UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Invalidating web applications attacks by employing the right secure code

Ricardo Jorge Graça Morgado

Mestrado em Informática

Dissertação orientada por:
Prof. Doutora Ibéria Vitória de Sousa Medeiros
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2019

Acknowledgments

In first place, I would like to thank my parents for motivating me to pursue this academic degree.

In second place, I would like to thank my advisors, professors Ibéria Medeiros and Nuno Neves, for their helpful advice during this dissertation and for constantly providing ideas on how I could make my work better. I would like to thank professor Ibéria for always keeping me informed about upcoming deadlines.

I would also like to thank my friends from the LASIGE Research Unit for all the fun conversations we had during the last few months.

This work was partially supported by the FCT through the project SEAL (PTDC/CCI-INF/29058/2017), and LASIGE Research Unit (UID/CEC/00408/2019).

To my family and friends.

Resumo

Desde o seu aparecimento, as aplicações *web* têm vindo a tornar-se cada vez mais populares e tornaram-se numa parte essencial das nossas vidas. Usamo-las todos os dias para fazer diversas tarefas tais como fazer compras, consultar o saldo da nossa conta bancária e entrar em contacto com os nossos familiares e amigos. Atualmente, as aplicações *web* são a forma mais utilizada para aceder aos serviços e recursos das organizações. No entanto, são conhecidas por conter vulnerabilidades no seu código-fonte. Estas vulnerabilidades, quando exploradas, podem causar danos severos às organizações como, por exemplo, o roubo de milhões de credenciais dos utilizadores e o acesso a informação confidencial, o que as torna num alvo apetecível para utilizadores mal intencionados. Por esta razão, é essencial que o acesso a serviços críticos tais como serviços de saúde e financeiros, seja feito através de aplicações *web* seguras.

A utilização de código seguro nas aplicações é de uma importância extrema para obter aplicações seguras e garantir a segurança dos seus utilizadores. As vulnerabilidades são deixadas inadvertidamente no código-fonte por programadores porque estes não têm o conhecimento necessário para escrever código seguro ou porque os testes de *software* não dedicam tempo suficiente à segurança. Por outro lado, os programadores que utilizam nas suas aplicações funções seguras da linguagem de programação acreditam que as suas aplicações estão protegidas. No entanto, algumas destas funções não invalidam todos os ataques e deixam as aplicações vulneráveis.

Este trabalho é focado na linguagem PHP porque esta é atualmente a linguagem de programação mais utilizada para o desenvolvimento de aplicações *web*. A linguagem PHP permite aos programadores realizarem ações que não seriam possíveis noutras linguagens, o que torna mais fácil aos programadores cometer erros. A linguagem PHP contém um grande número de funções seguras que podem ser utilizadas para remover vulnerabilidades dos diversos tipos. No entanto, uma grande maioria destas funções não é segura em todos os contextos ou é específica para um tipo de vulnerabilidade, o que cria a possibilidade de serem utilizadas incorretamente. Este problema torna mais fácil o aparecimento de vulnerabilidades se for tido em consideração o facto de uma grande parte dos cursos de programação existentes atualmente não dar ênfase suficiente à segurança. Por último, um outro fator que contribui para o aparecimento de vulnerabilidades é a complexidade das aplicações *web* atuais. Tal complexidade deve-se ao facto de as tecnologias disponíveis na

web terem sofrido uma evolução significativa nos últimos anos, o que leva ao aumento da quantidade de linguagens de programação e funcionalidades que os programadores têm de conhecer.

Atualmente, existe um grande número de ferramentas de análise estática destinadas a analisar código-fonte PHP e encontrar potenciais vulnerabilidades. Algumas destas ferramentas são baseadas em *taint analysis* e outras baseadas em análise dinâmica, execução simbólica, entre outras técnicas. Um problema conhecido destas ferramentas é o facto de, por vezes, reportarem vulnerabilidades que não são reais (falsos positivos), o que pode levar o programador a perder tempo à procura de problemas que não existem. Este tipo de ferramentas dá aos programadores relatórios em formatos variados e a esmagadora maioria delas deixa para o programador a tarefa de verificar se as vulnerabilidades reportadas são reais e removê-las caso o sejam. No entanto, muitas delas não dão informação sobre como remover as vulnerabilidades. Dado que muitos programadores estão mal informados acerca da escrita de código seguro, este processo nem sempre elimina as vulnerabilidades por completo.

Apenas um pequeno número de ferramentas de análise estática realiza a correção automática do código-fonte das aplicações e as que o fazem muitas vezes têm limitações. Destas limitações, destaca-se o facto de inserirem código sintaticamente inválido que impede o funcionamento correto das aplicações, o que permite a introdução de melhorias nesta área.

De entre os vários tipos de vulnerabilidades que podem ocorrer em aplicações *web*, os dois mais conhecidos são a injeção de SQL e o *Cross-Site Scripting*, que serão estudados em detalhe nesta dissertação.

Esta dissertação tem dois objetivos principais: em primeiro lugar, estudar estes dois tipos de vulnerabilidades em aplicações *web* PHP, os diferentes ataques que as exploram e as diferentes formas de escrever código seguro para invalidar esses ataques através da utilização correta de funções seguras; em segundo lugar, desenvolver uma ferramenta capaz de inserir pequenas correções no código-fonte de uma aplicação *web* PHP de modo a remover vulnerabilidades sem alterar o comportamento original da mesma.

As principais contribuições desta dissertação são as seguintes: um estudo dos diferentes tipos de ataques de injeção de SQL e *Cross-Site Scripting* contra aplicações *web* escritas em PHP; um estudo dos diferentes métodos de proteger aplicações *web* escritas em PHP e as situações em que os mesmos devem ser usados; o desenvolvimento de uma ferramenta capaz de remover vulnerabilidades de aplicações *web* escritas em PHP sem prejudicar o seu comportamento original; uma avaliação experimental da ferramenta desenvolvida com código PHP artificial gerado automaticamente e código PHP real.

A solução proposta consiste no desenvolvimento de uma ferramenta de análise estática baseada em *taint analysis* que seja capaz de analisar programas PHP simplificados e, caso estejam vulneráveis, inserir linhas de código com correções simples que removam tais

vulnerabilidades. Tudo isto sem alterar o comportamento original dos programas. A ferramenta desenvolvida limita-se exclusivamente à inserção de novas linhas de código, sem modificar as já existentes, para minimizar a probabilidade de tornar um programa sintaticamente inválido. Isto permite remover vulnerabilidades de aplicações *web* e, ao mesmo tempo, ensinar aos programadores como escrever código seguro. Os programas PHP simplificados que a ferramenta analisa consistem em ficheiros PHP contendo um único caminho do fluxo de controlo do programa original a que correspondem. Este programa simplificado não pode conter estruturas de decisão nem ciclos. A decisão de analisar programas simplificados foi tomada para permitir manter o foco desta dissertação na inserção de correções seguras, algo que atualmente apenas é feito por um pequeno número de ferramentas.

Para avaliar a ferramenta desenvolvida, utilizámos cerca de 1700 casos de teste contendo código PHP artificial gerado automaticamente com vulnerabilidades de *Cross-Site Scripting* e seis aplicações *web* reais, escritas em PHP, contendo o mesmo tipo de vulnerabilidade. Foram também utilizados 100 casos de teste contendo código PHP artificial com vulnerabilidades de injeção de SQL. A ferramenta conseguiu analisar todos os ficheiros PHP. Relativamente à capacidade de a ferramenta inserir correções no código-fonte das aplicações, obtivemos resultados encorajadores: todos os ficheiros que foram corrigidos continham código PHP sintaticamente válido e apenas um ficheiro viu o seu comportamento original alterado. O ficheiro cujo comportamento foi alterado apresenta uma estrutura mais complexa do que a esperada para um programa simplificado, o que influenciou a execução da nossa ferramenta neste caso.

Relativamente à capacidade de a ferramenta detetar vulnerabilidades, verificámos que a mesma reportou algumas vulnerabilidades que não são reais. Tal situação aconteceu em parte devido ao uso de expressões regulares nas aplicações *web*, algo que causa muitas dificuldades a ferramentas de análise estática. Verificámos também que muitos dos falsos negativos (vulnerabilidades reais que não foram reportadas) se deveram ao contexto em que determinadas funções seguras são utilizadas, algo que, mais uma vez, causa muitas dificuldades a ferramentas deste tipo. As situações referidas aconteceram principalmente no código artificial, que não deve ser visto como representativo de aplicações *web* reais. Assim, podemos afirmar que a nossa ferramenta lida eficazmente com código PHP real, o que abre a porta à possibilidade de a mesma ser utilizada para corrigir vulnerabilidades em aplicações disponíveis ao público.

Após esta avaliação experimental, concluímos que a solução desenvolvida cumpriu os objetivos principais para os quais foi concebida, ao ser capaz de remover vulnerabilidades sem prejudicar o comportamento original dos programas. A solução desenvolvida constitui uma melhoria nas capacidades das ferramentas de análise estática existentes atualmente, em especial das que realizam correção automática de código.

O estudo realizado acerca destes dois tipos de vulnerabilidades permitiu também obter

uma fonte de informação correta e confiável acerca das formas de escrever código seguro para prevenir os dois tipos de vulnerabilidades estudados em aplicações *web* escritas em PHP.

Palavras-chave: vulnerabilidades em aplicações *web*, análise estática, segurança de *software*, correção de código

Abstract

Currently, web applications are the most common way to access companies' services and resources. However, since their appearance, they are known to contain vulnerabilities in their source code. These vulnerabilities, when exploited, can cause serious damage to organizations, such as the theft of millions of user credentials and access to confidential data. For this reason, accessing critical services, such as health care and financial services, with safe web applications is crucial to its well-functioning.

Often, vulnerabilities are left in the source code unintentionally by programmers because they do not have the necessary knowledge about how to write secure code. On the other hand, programmers that use secure functions from the programming language in their applications, employing thus secure code, believe that their applications are protected. However, some of those functions do not invalidate all attacks, leaving applications vulnerable.

This dissertation has two main objectives: to study the diverse types of web application vulnerabilities, namely different attacks that exploit them, and different forms to build secure code for invalidating such attacks, and to develop a tool capable of protecting PHP web applications by inserting small corrections in their source code.

The proposed solution was evaluated with both artificial and real code and the results showed that it can insert safe corrections while maintaining the original behavior of the web applications in the vast majority of the cases, which is very encouraging.

Keywords: web application vulnerabilities, static analysis, software security, code correction

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, web applications have become increasingly popular and an essential part of our lives. We use them to perform everyday tasks such as checking our bank account balance, performing shopping within the comfort of our homes and contacting our friends and family. To support this functionality, web applications require us to provide them with our personal details, such as our home address and credit card number. These details are stored on the application's database and may be exposed if someone can successfully perform an attack.

Web applications often contain bugs that may reside undetected in their source code for many years. Some of these bugs evolve into security vulnerabilities when exploited by an attacker. By leveraging such vulnerabilities, attackers can steal or modify users' personal information, target them with advertisements or even shut down the application. Depending on the size and popularity of the application, a vulnerability might impact anything from a few dozen to millions of users and cause irreparable damage to a company's reputation. These factors make web applications an appealing target for attackers because, if they succeed, they can potentially compromise the personal details of up to millions of users.

Vulnerabilities are often caused by unaware and misinformed developers who make mistakes when writing their code. Programming courses often teach how to write code with little or no concern for security, thus building up bad habits within developers. There are also the problems of time and budget within organizations. Organizations often want their software developed within tight deadlines and with low cost, meaning that security is often not taken into consideration until it is too late. Software tests often value the user experience and system requirements over security because organizations want lots of people to use their applications. Lastly, the sources of information available to developers can sometimes have confusing information, which can also influence the security of the code [1] [6].

Thanks to advances in web technologies made over the last few years, web applications are now very complex and can even simulate the behavior of desktop applications.

This is achieved by resorting to various programming languages and frameworks. A web application typically includes a mixture of PHP, Javascript, HTML, CSS and SQL, amongst other languages. This mixture of languages and frameworks also puts an extra burden on developers, because they have to remember how to use each of them. Ultimately, this also contributes to the appearence of vulnerabilities because developers can get confused in the midst of so many languages.

All programming languages, namely PHP, contain a wide range of functions (and other methods) that can be used to remove vulnerabilities and invalidate attacks, however, most developers do not know how or when to use them. There are also many tools available to analyze applications and find potential flaws, however, such tools are often hard to use, do not provide the information developers need and report vulnerabilities that are not real (i.e., false positives).

Because most tools require that the developers correct manually the reported bugs, the existence of tools that can automatically detect and correct such problems would facilitate the developers' task of finding and removing vulnerabilities from their code. However, there are not many tools available with this capability and the ones that exist often have limitations in the sense that they produce syntactically invalid new programs that can not be executed.

1.1 Motivation

Given how important web applications are nowadays, it is of vital importance to ensure that their source code has no vulnerabilities. According to the OWASP Top 10 - 2017 [24], injection vulnerabilities (such as SQL Injection) rank number one and Cross-Site Scripting (XSS) ranks number seven in the list of web application security risks. Since they are prevalent and potentially have high impact, these are the types of vulnerabilities we will focus on this work, with XSS receiving most of our attention.

Our work is focused on PHP because it is currently the most used server-side programming language. As of June 2019, it powers around 79% of the websites whose server-side programming language is known[1]. PHP also powers Wordpress[2], Drupal[3] and Joomla[4], three popular Content Management Systems (CMS) that can be extended via custom modules and plugins made by the developers. These modules and plugins are known for being a rich source of vulnerable code. The fact that PHP is a "weakly-typed" language makes it easier to make mistakes in some situations, especially when dealing with badly-documented code. This happens because most Integrated Development Environments (IDEs) can not be sure of the type of data that a variable is going to have at

[1] https://w3techs.com/technologies/details/pl-php/all/all
[2] https://wordpress.com/
[3] https://www.drupal.org/
[4] https://www.joomla.org/

runtime in a given point in the program, thus providing less help to the developers in the form of errors and warnings.

Currently, there are several tools available to analyze PHP source code and detect potential vulnerabilities. Some of these tools are based on taint analysis and others on techniques such as dynamic analysis [18] and symbolic execution [4] [12] [26]. These tools provide reports in varying formats and almost all of them leave the burden of correcting the vulnerabilities to the developers. Given that most developers are unaware of the correct way to fix a given bug, this process often does not completely eliminate the vulnerability. Only a small number of these tools performs automatic correction of the application's source code, and the ones that do so often insert invalid fixes that break the expected behavior of the applications, thus leaving room for improvements.

This dissertation focuses on studying two prevalent types of web application vulnerabilities, namely XSS and SQL Injection (SQLi), as well as the attacks that exploit them and the ways of writing secure code for invalidating those attacks. In addition, it focuses on the development of a tool than can protect a PHP web application by inserting small fixes in the code that do not break the applications' functionality, thus improving the work performed by existing tools.

1.2 Objectives

The main objectives of our work are the following:

1. To study the various types of XSS and SQLi attacks against PHP web applications and to design an approach that could prevent such attacks by re-writing the code. A study like this would provide the community with a source of correct information regarding the security of PHP web applications;

2. To develop a tool capable of inserting small fixes in the web application's code that remove a vulnerability without breaking the application's original behavior. This would allow developers to correct their code and learn how they could have made it secure in the first place.

1.3 Contributions

The main contributions of this dissertation are the following:

1. A study of the different types of XSS and SQLi attacks against PHP web applications;

2. A study of the different sanitization methods available in PHP and the situations when they should be used;

3. Development of a tool capable of correcting PHP web applications while maintaining their original functionality;

4. An evaluation of the developed tool with both artificial and real web application code.

1.4 Document Structure

This document is structured as follows: Chapter 2 provides a description of the sanitization methods provided by PHP and when they should be used. It also includes an overview of other research efforts in the area of static analysis, with a special emphasis on taint analysis. Chapter 3 provides an overview of our solution to correct source code, the challenges it faces and how we solved them. Chapter 4 describes our solution's implementation in detail. Chapter 5 describes the evaluation we conducted of our solution. Lastly, Chapter 6 presents our conclusions and directions for future work.

Chapter 2

Context and Related Work

This chapter provides some context and gives an overview of related work. It describes the vulnerabilities for our work and explains how they can be attacked, following with a description of the PHP sanitization functions that can prevent such attacks and the situations in which they should be applied. Lastly, we will briefly present some tools that employ static analysis techniques, with a special emphasis on those that perform automatic code correction.

2.1 Vulnerabilities

The IETF RFC 4949 [21] defines a vulnerability as: "*A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy*." As the definition clearly states, not all software flaws can be considered vulnerabilities. Only those that can be exploited to violate the system's security policy can be considered as such. We decided to focus our work on two types of vulnerabilities: XSS and SQLi. We made this decision because these are the two most well-known types of vulnerabilities in web applications and they are considered as very prevalent by the OWASP Top 10 - 2017 [24], making them a good starting point for the development of the solution we propose.

2.1.1 A1: Injection

Injection flaws, such as SQL and OS, occur when untrusted data is sent to an interpreter as part of a query or command [24], repectively, thus tricking the interpreter into executing unintended commands. The most well-known type of injection flaw is SQL Injection (SQLi), in which an attacker sends an especially crafted input to a database as part of a query. This allows the attacker to bypass authentication mechanisms, gain access to confidential information, insert or modify data or even shut down the database server. SQLi flaws are often introduced by developers who fail to properly sanitize their input data before inserting it into a query. This type of vulnerability can be prevented by applying

proper sanitization to the input and by using parameterized queries to interact with the database, amongst other techniques.

```
1 <?php
2 $query = "SELECT * FROM Employee WHERE id = " . $_GET["id"];
3 $result = mysqli_query($conn, $query);
```

Listing 2.1: Example of a classic SQLi vulnerability in PHP.

Listing 2.1 shows an example of a piece of code vulnerable to SQLi. In this example, the input obtained via the `$_GET` array is included directly in the query, without any form of validation or sanitization. In Listing 2.2 there is an example of an input that triggers the vulnerability. This input (`1 OR 1=1`) causes the conditional part of the query to always evaluate to true (referred to as a tautology [11]), when executed by `mysqli_query`, thus leading to the return of more records than the developer originally intended. This could lead to the exposure of confidential information, if `$result` is for instance returned to the user.

```
1 <?php
2 $query = "SELECT * FROM Employee WHERE id = 1 OR 1=1";
3 $result = mysqli_query($conn, $query);
```

Listing 2.2: Example of a SQLi vulnerability exploitation (the attacker's input is underlined).

2.1.2 A7: Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) flaws occur when an application includes untrusted data as part of a web page without proper validation or escaping [24]. This type of flaws allows an attacker to trick the victim's web browser into executing malicious code. There are three types of XSS attacks:

Reflected XSS: This type of attack occurs when the user's input is immediately returned by the web application without being made safe to render in the browser. This type of attack requires the victim to be tricked into, for example, clicking on a link before it can succeed. Listing 2.3 shows an example of Reflected XSS. As described before, the input obtained via the `$_GET` array is directly "reflected" on the resulting web page after being used in an `echo` statement.

```
1 <?php
2 $user = $_GET["user"];
3 echo "Welcome, " . $user;
```

Listing 2.3: Example of a Reflected XSS vulnerability in PHP.

Stored XSS: This type of attack occurs when untrusted input is stored on the server and later sent to a victim without being made safe to render in a browser. This type of

attack requires no action from the victim and it can have a higher impact due to the larger number of potential victims involved. In Listing 2.4, there is an example of this type of XSS. A query (defined in the `$query` variable) is executed in line 3, via the call to `mysqli_query`. The result of this execution is stored in `$result` and consists of an object of type `mysqli_result`, that contains all rows returned by the query. In line 4, a while loop is used to iterate over all rows contained in `$result` via the `mysqli_fetch_array` function. Each time this function is called, it returns one row from `$result` in the form of an array. In each iteration of the while loop, the variable `$row` contains one row from `$result`. In line 5, the value of position 1 of `$row` is included in a call to `echo`, thus being sent to the victim as part of the output. If this valued contained Javascript, that code would be sent to the victim's browser and executed.

It is important to note that, for the attack to succeed, the attacker would need to insert previously an especially crafted message in the application's database via some of it's input.

```
1 <?php
2 $query = "SELECT * FROM Suggestion";
3 $result = mysqli_query($conn, $query);
4 while ($row = mysqli_fetch_array($result))
5     echo "Message: " . $row[1];
```

Listing 2.4: Example of a Stored XSS vulnerability in PHP.

DOM-based XSS: This type of attack occurs when the entire untrusted data flow takes place in the victim's browser. To succeed, it requires developers to use unsafe Javascript functions, such as `eval()` and `document.write()`. Listing 2.5 shows an example of this type of XSS. In this example, the content of the current URL (`document.location`) is written onto the HTML by a call to `document.write()` made in a client-side script, which means that users tricked into opening the web page via a URL such as

`http://example.com/index.php#<script>alert(1)</script>`

will be attacked.

The attack occurs because the result of calling `decodeURIComponent` is written onto the HTML by `document.write()`, a function that does not encode HTML's special characters. The `decodeURIComponent` function URL-decodes the string it receives as argument. In this case, `document.location` contains a URL-encoded version of the current web page's URL (note that the URL contains some Javascript in it). This means that the call to `decodeURIComponent` made in line 3 returns a value that contains some valid Javascript in it. Because this value

is written onto the HTML by `document.write()`, it results in the inclusion of a `<script>` tag in the HTML document. The browser then executes this script, allowing the attack to succeed.

It is important to note that the server had no influence on this XSS attack because the attacker's payload was included after the # character and was not sent to the server.

```
1 <script>
2     document.write("Current URL " +
3             decodeURIComponent(document.location));
4 </script>
```

Listing 2.5: Example of a DOM-based XSS vulnerability.

The reflected and stored variants of XSS can be prevented by properly escaping potentially malicious input before including it in a web page. There are also other techniques that can be used, capable of preventing XSS in all web applications, regardless of their programming language. Such techniques include defining a Content Security Policy (CSP)[1] or defining the `X-XSS-Protection` HTTP header[2]. The CSP can prevent all variants of XSS while the `X-XSS-Protection` header can only prevent the reflected variant of XSS.

To prevent the DOM-based variant of XSS in Javascript, developers need to make use of safe functions and attributes, such as `innerText` and `textContent`.

2.2 Sanitization Methods

2.2.1 Generic Sanitization Methods

This subsection presents some sanitization methods available in PHP. They are referred to as generic because they can be used to prevent both XSS and SQLi attacks.

For numeric inputs, the PHP language contains two functions that can prevent many of the attacks against the two types of vulnerabilities being considered in our work (except DOM-based XSS). These functions are `intval` and `floatval`. Both functions receive a string as their argument and return the result of converting that string to an integer and a float, respectively. If they are unable to convert the string to a number, they return zero, thus making any malicious input harmless while leaving benign inputs untouched. In addition to the two aforementioned functions, there is the possibility of using casts to numeric types. The casts will perform in the same way as the previously described functions, thus making inputs harmless.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

For string inputs that have a well-known format, such as a date or a zip code, there is the possibility of using the `preg_match` function to compare it with a regular expression. In order for this technique to be safe, the developer has to use a correct regular expression.

Lastly, if the input can only assume one of a limited number of values, developers can use white lists to ensure it is safe. There are many ways to do this in PHP, but one possibility is to create an array of valid values and use the `in_array` function to verify if the input is part of the array.

2.2.2 Cross-Site Scripting Sanitization

The reflected and stored variants of XSS can be prevented by using functions that encode special characters, making them harmless to render in the browser. There are two groups of functions that can be used:

HTML-encoding functions: These functions convert all of HTML's special characters to their respective representation in HTML entities, thus preventing attacks that abuse unintended utilization of these values. For example, the < character is converted to `<` , thus being displayed on the screen by the browser. There are two functions in this group: `htmlspecialchars` and `htmlentities`. They receive the same arguments (a string to be sanitized and a set of flags), but differ in the number of characters they convert. The difference between them lies in the number of characters they convert. `htmlspecialchars` converts the following characters: `&`, `"`, `'`, <, >. `htmlentities` converts these characters and more than two hundred additional ones, such as `Á` and `Ç`.

Despite their difference, both of them encode all of HTML's special characters, making their input safe to render in the browser in most situations. The used flags can influence the safety of these functions because the flags specify the way the function encodes quotes and the way it deals with the HTML itself (whether it considers the HTML as HTML 4.01 or HTML 5, for example).

URL-encoding functions: These functions perform URL-encoding of their input, meaning that any non-alphanumeric character is encoded and made harmless to render in the browser. As an example, the < character is converted to `%3C` , thus being made harmless, as the browser will display `%3C` on the screen instead of the actual character. There are three functions in this group: `urlencode`, `rawurlencode` and `http_build_query`. `http_build_query` receives an array and returns a URL-encoded query string with all key-value pairs contained in the array. The other two functions receive a single string as an argument and differ only in the way they encode spaces. `urlencode` encodes spaces as + while `rawurlencode` encodes spaces as `%20`. Functions belonging to this group will make their input harmless in all situations, but they may cause the input to be displayed on the screen in it's

Encoding Function	Returned Value	Rendered by the browser as
`htmlentities`	<script>	<script>
`urlencode`	%3Cscript%3E	%3Cscript%3E

Table 2.1: Comparison between the result of the two forms of encoding, and how they are rendered in the browser.

URL-encoded form. For this reason, these functions should only be used in some special situations.

To illustrate the differences between the two forms of encoding, we provide an example in Table 2.1. In this table, both forms of encoding receive as input the string `<script>`. Regarding the DOM-based variant of XSS, it can only be prevented in Javascript, which is not the main focus of our work.

2.2.3 SQL Injection Sanitization

SQL Injection attacks can be prevented by using functions of the `*_escape_string` family. These functions escape SQL's metacharacters in order to make their input safe to include inside of a SQL query string. For MySQL, the `mysql_real_escape_string` and `mysqli_real_escape_string` functions can be used to prevent SQLi in some situations. Both functions escape the same characters in the same way. The difference between them lies in the PHP extension they use. The former uses the MySQL[3] extension while the latter uses the MySQL Improved extension.

Other than these functions and the generic solutions, the safest way to prevent SQLi is to use prepared statements. Prepared statements[4] consist of two phases: preparation and execution. In the preparation phase, the statement is sent to the database server, which performs a syntax check and initializes resources for later use. In the execution phase, the client binds parameter values and sends them to the server. The server then executes the statement with the bound values using the previously initialized resources. This ensures that user-supplied values are never treated as SQL commands. In PHP, the `mysqli_prepare` function allows the creation of prepared statements. After creating the statement, the developer has to bind values to all of it's parameters and execute it. To do this, the functions `mysqli_stmt_bind_param` and `mysqli_stmt_execute` are available.

The use of prepared statements is demonstrated in Listing 2.6. In this example, a string containing SQL code is assigned to the variable `$query` in line 2. Note that this string contains a question mark in the location where the input is going to be bound later. Next, in line 3, a call to `mysqli_prepare` is made to execute the preparation phase. This call

[3]The MySQL extension was deprecated in PHP 5.5 and removed in PHP 7.
[4]Prepared statements are not available in the MySQL extension.

returns an object of type `mysqli_stmt`. In line 4, the value of `$_GET["id"]` is bound to the statement (in the location where the question mark is). The string `"i"` given as the second argument to `mysqli_stmt_bind_param` specifies that the argument should be treated as an integer. The prepared statement (with a value bound to it's only parameter) is executed in line 5 and the result of this execution is retrieved in line 6.

```
1 <?php
2 $query = "SELECT * FROM Employee WHERE id = ?";
3 $stmt = mysqli_prepare($conn, $query);
4 mysqli_stmt_bind_param($stmt, "i", $_GET["id"]);
5 mysqli_stmt_execute($stmt);
6 $result = mysqli_stmt_get_result($stmt);
```

Listing 2.6: Use of prepared statements in PHP.

2.2.4 PHP Filters

PHP filters can be employed as a sanitization method by calling the `filter_var` function, and by providing as it's second argument a constant that identifies the filter to be used. This sanitization method can operate as both a generic one and a XSS one, depending on the provided filter. The sanitization filters[5] available are all named `FILTER_SANITIZE_*`. These filters range from number sanitization that act similarly to `intval` to HTML-encoding that performs the same job as `htmlspecialchars`. Examples of such filters are `FILTER_SANITIZE_NUMBER_INT` and `FILTER_SANITIZE_SPECIAL_CHARS`, respectively. There are also filters that perform URL-encoding. Note that the only filters that can prevent SQLi are the ones that operate as the generic sanitization methods.

2.3 Safety of Sanitization Methods

2.3.1 Pitfalls of Cross-Site Scripting Sanitization

The HTML-encoding functions can prevent the stored and reflected variants of XSS when their result is included inside the content of any HTML tag, other than the `<script>` and `<style>`[6] tags. They will allow attacks to go through when their result is included in an unquoted part of any HTML tag's definition. They will also fail if they are called without the `ENT_QUOTES` flag (or with the `ENT_NOQUOTES` flag) and their result is included inside of a string quoted with single quotes. The most widely recommended way to call these functions safely is to use solely the `ENT_QUOTES` flag. Listings 2.7 and 2.8 show examples of safe and unsafe usage of `htmlentities`, respectively. Listing 2.9 presents an example of an attack against the code in Listing 2.8.

[5] https://www.php.net/manual/en/filter.filters.sanitize.php

[6] The execution of Javascript inside this tag is only possible in older browsers.

```php
<?php
$input = htmlentities($_GET["a"], ENT_QUOTES);
echo "<p>" . $input . "</p>";
```

Listing 2.7: Safe usage of `htmlentities`.

```php
<?php
$input = htmlentities($_GET["a"]);
echo "<body bgcolor='#" . $input . "'>Hello World!</body>";
```

Listing 2.8: Unsafe usage of `htmlentities` (called without flags).

```php
<?php
$input = htmlentities($_GET["a"]);
echo "<body bgcolor='000' onload='alert(1)'>Hello World!</body>";
```

Listing 2.9: Example of an attack against the code in Listing 2.8 (the attacker's input is underlined).

The URL-encoding functions can prevent the stored and reflected variants of XSS in all situations because they encode all non-alphanumeric characters. This means that it is not possible to write meaningful Javascript if the input goes through one of these functions. However, the use of HTML-encoding is preferred over URL-encoding in most situations, as the URL-encoding functions cause the input to appear in it's URL-encoded form, which may cause the application to be less appealing to the user, as shown in Table 2.1. URL-encoding should only be used in the situations when HTML-encoding is unsafe.

2.3.2 Pitfalls of SQL Injection Sanitization

Functions of the `*_escape_string` family can only prevent SQLi if their result is included inside of a SQL string. This happens because they only sanitize characters that can influence a string's limits, such as quotes and line breaks. For example, if their result is included in a query, in a comparison with an integer, they will let the attack proceed. Listings 2.10 and 2.11 show examples of safe and unsafe usage of these functions, respectively.

```php
<?php
$query = "SELECT * FROM Employee WHERE name = '" .
          mysqli_real_escape_string($conn, $_GET["name"]) . "'";
$result = mysqli_query($conn, $query);
```

Listing 2.10: Safe usage of the string escaping function.

```php
<?php
$query = "SELECT * FROM Employee WHERE id = " .
            mysqli_real_escape_string($conn, $_GET["id"]);
$result = mysqli_query($conn, $query);
```

Listing 2.11: Unsafe usage of the string escaping function (`id` is an integer).

In Listing 2.11, the usage of `mysqli_real_escape_string` is unsafe because the location where the input is included in the query is not delimited by quotes. This means that because this function does not modify inputs such as `1 OR 1=1` (that contain no quotes), it allows them to be included as part of the query's structure, thus being executed by the database server. This means that these functions can only correct a small subset of the possible SQLi vulnerabilities, meaning that developers should resort to other sanitization methods when these functions are unsafe.

It is important to note that prepared statements do not work in all situations. Prepared statements do not allow the binding of parameters to table or column identifiers or SQL keywords meaning that, in this situation, developers should resort to white lists to validate their input against a set of values known to be valid (the study performed in [3] suggests that this situation is uncommon). Prepared statements also do not work if developers make incorrect use of them, which is likely to happen given that they are more complex than simple sanitization functions.

2.4 Static Analysis

Static analysis has the objective of analyzing the source code of an application to find bugs. Since some bugs may later evolve into vulnerabilities, this technique is of vital importance for developers. Static Analysis Tools (SATs) analyze the whole code without executing it. This allows developers to use them in any stage of the application development process, even if their application is not complete. To achieve their objective, SATs compare the code to a set of patterns that indicate a vulnerability. If the tool's knowledge base does not contain a pattern for a given type of vulnerability, the tool will not report it, leading to a false negative.

When compared to manual code auditing, SATs offer considerable advantages because they can analyze much larger code repositories in a relatively short time. On the other hand, they are not as accurate as a human being at finding vulnerabilities and may report false positives (reporting a flaw that is not actually a vulnerability). False positives are particularly bad because they cause the developer to waste time looking for unexistent problems. Most SATs still require human intervention to verify that the bugs reported are in fact vulnerabilities, and fix them when necessary.

Several authors have conducted studies on the way developers interact with SATs. Oyetoyan et al. [17] conducted a study in which they aimed to learn what developers expect from SATs. They learned that developers prefer tools that are easy to install and use, provide easy-to-understand messages and provide no false positives. Smith et al. [22] conducted a study in which they equipped developers with a static analysis tool and observed their interactions with security vulnerabilities in an open-source system they had contributed to. They concluded that developers often do not use SATs because such

tools do not provide information aligned with their needs. The authors also observed that developers would benefit from program flow navigation tools while investigating vulnerabilities.

One of the forms of static analysis is taint analysis. Taint analysis consists of an approach in which program variables receive one of two labels: tainted (their value is potentially unsafe) and untainted (their value is safe). In this technique, the SAT tracks the status of a program's variables from their definition to a sensitive sink (function that expects to receive untainted data). The locations where user-controlled input is assigned to variables for the first time are referred to as entry points. Initially, all variables that receive user-controlled input are marked as tainted. When the value of a tainted variable is assigned to an untainted variable, the former's taint status is "propagated" to the latter, in a process known as taint propagation. If a tainted variable is passed through a safe sanitization function, it is marked as untainted. The main difficulty of taint analysis lies in this mechanism of marking variables as untainted because it is often difficult to decide whether a sanitization function is safe in a given context. Such decision often requires knowledge about the application's execution environment and internal structure. When faced with a decision like this, SATs have to make one of two choices: 1) mark the variable as tainted and possibly report a false positive or 2) mark the variable as untainted and possibly miss a vulnerability.

Halfond et al. [10] developed a novel form of taint analysis that they referred to as positive tainting. Their technique is based on the marking and tracking of trusted data, instead of untrusted data. In their work, they argued that traditional taint analysis can sometimes trust data that should not be trusted, leading the analysis to miss vulnerabilities and that positive tainting fails in a way that maintains the security of the system, because any input that is not explicitly trusted is considered to be unsafe. They believe that enumerating all sources of untrusted input for traditional taint analysis is error prone and enumerating all sources of trusted input is straightforward. Their approach eliminates the problem of false negatives that result from the incomplete identification of untrusted data sources. Their work, however, is focused solely on SQL Injection.

Dashe and Holz [5] developed an interesting approach that detects second-order vulnerabilities in web applications. Second-order vulnerabilities (such as stored XSS) occur when a payload is stored on the server and later used in a security-critical operation. This type of vulnerability is often introduced by developers who believe that data coming from the application's data storage is safe. The authors developed an approach that identifies taintable data stores and checks if a symbol originating from such a data store reaches a sensitive sink without being sanitized.

Livshits and Lam [14] developed a static analysis approach that is based on context-sensitive pointer alias analysis and introduced extensions to the handling of strings and containers to improve the precision. Their approach, however, requires a developer-

provided specification written in a language similar to Java in order to find vulnerabilities. Using this specification, they can find sink objects derivable from source objects. If the developer-provided specification is incomplete, their technique may fail to report some vulnerabilities.

AMNESIA [9] is a tool that protects web applications in two phases: a static phase and a dynamic phase. In the static phase, the tool uses string analysis to extract a model of all the queries that could be generated from the application's source code. The creation of this model is based on the intuition that the application's source code implicitly contains a "policy" of legitimate queries. In the dynamic phase, the application is instrumented to check the actual query against the previously created model. If a query does not match the model, it is prevented from executing on the database.

In recent years, there have also been some research efforts focused on applying Machine Learning (ML) approaches to the detection of vulnerabilities in source code [8] [25] [19] [20]. Yamaguchi et al. [25] developed an approach that allows developers to find unknown vulnerabilities based on the code pattern of a known one. Their approach is based on the application of ML techniques to automatically determine patterns of API usage. They can then find functions with similar API usage patterns to the vulnerable one. Using this approach, they were able to discover an unknown vulnerability in a popular library.

Flynn et al. [7] developed and tested classification models that predict if static analysis alerts are true or false positives, using a combination of multiple SATs. Their results showed that accurate classifiers were developed. In their work, they argued that it is necessary to prioritize alerts in order to repair all code flaws within the project's budget.

PhpMinerI [19] and PhpMinerII [20] are two tools that use data mining to predict vulnerabilities. The data miners are learned from code patterns that represent input sanitization. This requires a set of annotated excerpts of code. PhpMinerII also applies dynamic analysis, in which the code of user-defined functions is executed with a predefined set of inputs in order to increase the analysis' accuracy.

Medeiros et al. [15] developed WAP, a static analysis tool for PHP web applications. The most novel aspects of WAP are: 1) the use of data mining to predict false positives and 2) automatic code correction. WAP's taint analyzer uses taint analysis to find several types of vulnerabilities in the application's source code. The vulnerabilities are then processed by the data mining component to classify each one as a false positive or not. Lastly, the vulnerabilities that were not classified as false positives are corrected automatically. The taint analysis performed by WAP is global, interprocedural and context-sensitive. This allows WAP to perform a better analysis across multiple function calls in possibly different modules.

Nunes et al. [16] studied the problem of combining multiple SATs to detect web application vulnerabilities. They concluded that combining the results of multiple SATs has benefits due to the complementarity of the produced results. On the other hand, they also

concluded that, as the number of tools increases, both the vulnerabilities found and the false positives reported increase. Algaith et al. [2] also studied the problem of combining multiple SATs. They presented their results in terms of sensitivity (which measures the performance of the SAT to find vulnerabilities) and specificity (which measures the performance of the SAT to not raise false alarms), two well-established measures for binary classifiers. They tested several combinations of SATs and concluded that configurations that provide more specificity usually have lower sensitivity, meaning that an improvement in one of the measures implies sacrificing the other one.

2.4.1 Automatic Code Correction

Despite the large number of SATs available to find vulnerabilities in the source code of web applications, not many perform automatic code correction. In this subsection, we will look into two that do so.

WebSSARI [13] is a tool that statically analyzes existing code without requiring any additional effort from the developers. WebSSARI does, however, allow developers to add annotations to their source code to reduce the runtime overhead. Through static analysis, WebSSARI locates the portions of code requiring fixes and inserts guards to secure it. The insertion is made right after the statement that caused the variable to become tainted. The authors, however, do not explain how the guards are inserted or what they consist of.

WAP [15] applies corrections that follow common secure coding practices. The main downside of WAP is the fact that it sometimes causes the resulting program to become syntactically invalid, meaning that it can not be executed. Another downside of WAP is the fact that it inserts it's own corrections, meaning that developers have to include a special PHP file in their code to allow the corrections to work.

Chapter 3

Proposed Solution

In this chapter, we describe the main challenges that our proposed solution faces, and some decisions that we took to deal with them. We also present an overview of our proposed solution and it's main components. Lastly, we present the algorithm for automatic code correction in the form of pseudocode.

3.1 Design Challenges

In order for an automated tool to correct a web application, it must face some challenges that will be described in detail in this section. We will also describe the decisions we made to tackle these challenges.

3.1.1 Where to insert the correction?

It is hard for an automated tool to decide where to apply a correction in the code of a web application. Some variants of XSS and SQLi can be corrected by using a sanitization function to make the input safe to include in the HTML or a SQL query, respectively. However, even in this case, it is often difficult to determine where to place the sanitization function call without breaking the application logic.

```
1 <?php
2 $a = $_GET["v"];
3 echo "<div>" . $a . "</div>";
```

Listing 3.1: Example of a reflected XSS vulnerability in PHP.

Taking as an example the program program of Listing 3.1, the `htmlentities` function could be used to prevent the exploitation of the XSS vulnerability, by replacing line 2 with `$a = htmlentities($_GET["v"], ENT_QUOTES);`.

This would in fact remove the bug, but the `htmlentities` function might modify the length of the input string if it has to replace some characters with their HTML entity equivalents. If this program contained some additional logic based on the length of

that string, this modification could cause the application to behave erroneously. Listing 3.2 contains an example of a program in this situation. Assume that the call to `htmlentities` made in line 2 was added by an automated tool. Considering the input string `<script>alert(1)</script>`, the program's control flow should enter the "then branch" of the if statement because the input is exactly 25 characters in length. However, the call to `htmlentities` converts the input into it's representation in HTML entities (`<script>alert(1)</script>`), a string that is 37 characters in length. This causes the if statement's condition to evaluate to false, leading to a change in the program's behavior.

```
1 <?php
2 $a = htmlentities($_GET["v"], ENT_QUOTES);
3 if (strlen($a) <= 25) {
4     echo $a . " is a short string";
5 } else {
6     echo $a . " is a long string";
7 }
```

Listing 3.2: Example of a PHP program with logic based on the length of the input.

For this reason, an automated tool can not fix every application by applying the sanitization function to the entry point. However, adding the correction closer to the sensitive sink is more difficult than it seems because there are no guarantees that the source code's formatting will be easy to process by the tool. This can be a problem especially if the source code contains instructions that span over multiple lines or multiple instructions per line. In Listing 3.3, there is an example of a program containing an `echo` statement that spans over multiple lines. This makes the program hard to fix for an automated tool because adding a call to a sanitization function directly on the `echo` requires that function call to span over multiple lines as well. On the other hand, adding a new line to the program containing a function call requires the tool to obtain the whole of the array's key, which also spans over multiple lines.

```
1 $prefix = "pre";
2 $suffix = "suf";
3 echo $_GET[$prefix
4             .
5           $suffix];
```

Listing 3.3: Example of a PHP program with an instruction spanning over multiple lines.

```
1 <div>Welcome</div>
2 <?php $a = $_GET["v"]; ?>
3 <?php $q = "SELECT * FROM T WHERE id = " . $a; ?>
4 <div>Query results:</p>
5 <?php $result = mysqli_query($con, $q); ?>
6 <?php print_r($result); ?>
```

Listing 3.4: Example of a SQLi vulnerability in PHP.

Another example can be observed by considering the program in Listing 3.4, where the vulnerability could be fixed by replacing the PHP instruction on line 4 with

```
$q = "SELECT * FROM T WHERE id = " . intval($a);
```

It is important that the remainder of line 4 is not modified because, if so, it will cause the program to become syntactically invalid by modifying a PHP tag. Although this seems easy to the human eye, it is non-trivial for an automated tool to reason about what parts of a line of code it can modify safely.

This problem becomes even more complicated if the required correction requires more than the addition of a simple sanitization function. Such cases include for instance the use of prepared statements to correct SQLi vulnerabilities. In order to add a prepared statement to an existing program, the automated tool has to ensure that the statement (and it's multiple function calls) is added after the program's connection to the database is open, otherwise the application will completely stop working.

In order to solve this challenge, we decided to ensure that all corrections inserted by our approach consist solely of adding lines of code, instead of modifying existing ones. This will help us minimize the chances of making a program syntactically invalid. We also decided that our solution would always insert corrections that do not require the inclusion of new files in the application. To avoid breaking the application's logic by modifying the input string too early in the program, we decided to insert the correction for a given variable on the line of code immediately before the sensitive sink, if possible. This may not be possible in every program because some variables can not be sanitized in their entirety (for example, a variable that contains some developer-provided HTML can not be converted to HTML entities because that would produce an output different from what the developer intended). When a tainted variable `V` can not be sanitized on the line of code before the sensitive sink, we will sanitize the variable(s) that caused `V` to become tainted, in the line(s) of code immediately before that happened. We also propose that our approach would need to simulate the execution of the variable operations contained in the program. This would allow us to know the approximate value of a variable when it reaches a sensitive sink, thus knowing whether we can correct the variable in it's entirety.

3.1.2 What correction to insert?

The type of correction to insert in a web application is very closely related to the type of vulnerability contained in it. For XSS, most variants (except DOM-based) can be corrected by using the `htmlentities` or `htmlspecialchars` functions. For SQLi, the available corrections are more complex. If, for example, the input data is expected to be an integer, the `intval` function will protect the application. If the expected type of the input data can not be correctly determined, the safest option is to replace the existing query with a prepared statement.

However, there are some cases when the expected input is a string in a well-defined

format, such as a date. In these cases, functions such as `intval` can not be used but there is the possibility of employing regular expressions to validate the input.

These problems make it very difficult to determine the type of correction to apply, especially in the case of SQLi. For example, always inserting prepared statements will add an extra performance burden to the application, which is unnecessary in most cases, and it will also be harder to do due to the added amount of lines of code to insert. As mentioned before, if the user's input is included in a table or column identifier, the use of prepared statements will cause the application to stop working.

Determining the type of correction to apply requires the automated tool to be capable of reasoning about where the input data is inserted and what is it's expected type. This requires the automated tool to be capable of understanding how the query or HTML is constructed which can be difficult in some applications with lots of assignments distributed over the control flow.

To solve this challenge, we select the correction to apply based firstly on the type of vulnerability contained in the code. If it is SQLi, we will insert a string escaping function. If it is XSS, we will insert URL-encoding functions in the situations when they are the better option and HTML-encoding functions in all other cases.

3.1.3 How to deal with existing sanitization?

Dealing with existing sanitization or validation functions poses another problem because they might not be enough to prevent all attacks. In such cases an automated tool has to decide between making some modifications to the existing sanitization or adding it's own sanitization to the program. In PHP, something as simple as using a wrong flag in the `htmlentities` function is enough to leave an application vulnerable to XSS attacks. If an existing application uses `mysqli_real_escape_string` to escape an input that is later included in a SQL query, in a comparison with an integer column, the application will still be vulnerable to SQLi (this is showcased in Listing 2.11). Situations such as the ones described may cause the application to appear secure from the tool's point of view. The former can be corrected by replacing the wrong flag with a correct one, while the latter can only be fixed if the tool correctly determines the column's data type. If the decision is made to add new sanitization to the program, the solution must ensure that the inserted sanitization does not interfere with the existing one, which leads back to the problem of where the correction has to be inserted.

To tackle this challenge, we decided that our proposed solution would need to have the capability of dealing with diverse sanitizations methods. If the sanitization method in use by the application is safe, the variable is marked as untainted, meaning that no correction is applied at all. If the sanitization method is unsafe, a correction will be applied following the ideas described in Subsection 3.1.1. In some cases, the application might contain a safe sanitization method that is regarded as unsafe by our solution. In these cases, the

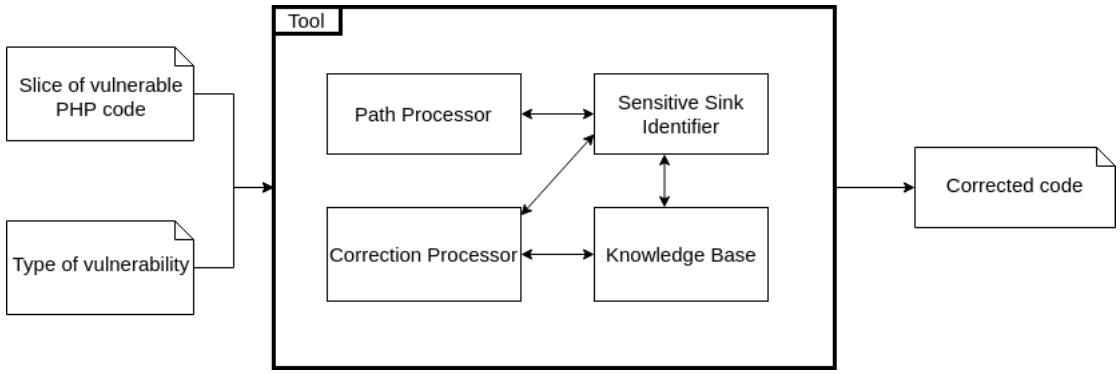

Figure 3.1: Overview of our solution's main components.

ideas described in Subsection 3.1.1 should help us to prevent our correction from breaking the application's logic.

3.2 Solution Overview

In this section, we will provide a description of our proposed solution, it's main components and it's input and output. Our proposed solution is based on taint analysis. Figure 3.1 provides an overview of our solution and it's components. The proposed solution receives the following input:

Slice of vulnerable PHP code: This is a PHP file containing a simplified version of a PHP program. The simplified program contains one control flow path that takes some input from an entry point (EP) to a sensitive sink (SS). This version of a program must not contain decision statements or loops. The slice can contain more code than the one relevant for the vulnerability as long as it contains a single control flow path. In addition, the path from the EP to the SS must not be inside the body of any function or method. The slice of code can contain multiple vulnerabilities as long as they are all of the same type. This PHP file is not required to be vulnerable.

Any other files included by the slice of code will be ignored by the solution.

We decided that our solution's input would consist of this type of file because it allows us to focus our efforts on the development of an efficient method to insert corrections in PHP code, without having to focus too much on the development of an efficient static analysis approach. We believe that the main limitations of existing solutions (that perform code correction automatically) lie in the insertion of corrections and not in the static analysis itself.

Type of vulnerability: This is simply a string containing the type of vulnerability to be analyzed in each execution of the solution. Currently, only XSS and SQLi are supported. Our proposed solution requires this as input because this allows our taint

analysis to be more efficient. With this, the solution can know from the start what is the type of vulnerability contained in the program and what type of sanitization functions should be considered safe.

The solution's output consists of a PHP file containing the corrected version of the slice of code provided as input. If no corrections are applied, no output file is returned due to the fact the the original code was not changed.

As mentioned before, the solution uses taint analysis to find vulnerabilities. It contains the following main components:

Path Processor: This is the component that performs the taint analysis itself and simulates the execution of variable operations. It is responsible for tracking the input's taint status from the entry points to the sensitive sinks and maintaining the status of the program's variables. In order to perform these tasks, it needs information provided by the Knowledge Base and the Sensitive Sink Identifier.

This component is needed due to the fact that our solution is based on taint analysis and needs to simulate the execution of variable operations. We decided to leave these tasks in a separate component as opposed to leaving them in for instance the Correction Processor because we wanted the taint analysis and code correction to be independent from each other.

Correction Processor: This is the component that determines the variable(s) that require correction, what corrections they require and the line(s) of code where those corrections should be applied. This is done using the information produced by the remaining three components. This component is also responsible for producing the actual line(s) of code to be inserted in the output file.

This component was created to ensure that the code correction is independent from the taint analysis.

Knowledge Base: This is the component that contains the names of all entry points, sensitive sinks and sanitization methods considered by our approach for each type of vulnerability. It is used by the Path Processor to obtain the names of entry points and sanitization methods. It is also used by the Sensitive Sink Identifier to obtain the names of the sensitive sinks.

We created a separate component to keep this information because we want it to be easily extendable in the future, to for instance add new sanitization functions without affecting the remaining components.

Sensitive Sink Identifier: This component is responsible for finding the sensitive sinks contained in the input program for all types of vulnerabilities supported by our

approach. It is used by the Path Processor and the Correction Processor to check whether a given instruction of the input program is a sensitive sink.

We decided to create a separate component to perform this task because we wanted the identification of sensitive sinks to be independent from the taint analysis and code correction.

3.3 Algorithm

In this section, we will describe the main ideia behind the algorithm we propose to correct PHP files and present it in the form of pseudocode.

3.3.1 Main Idea

The main idea behind our approach is the simulation of the operations involving PHP variables statically. This is done while the taint analysis is being performed and consists of simulating the execution of string concatenations and arithmetic operations involving numbers. When a value that can not be known during static analysis (such as a value obtained from an entry point) is involved in one of these operations, a special marker is inserted in it's place, thus indicating that part of the result contains some user input. The reason behind this idea is to know where the user input is inside of a string. This can be useful to determine if a string contains developer-provided HTML, in the case of XSS, or to determine where the input is being included in a query, in the case of SQLi. However, it is important to note that this simulation can not obtain the exact value of a variable for certain cases, such as function calls and arrays. Despite this, we belive that this approach can obtain an approximate value for a variable, especially when we consider that our input is a slice of code (i.e., a simplified PHP program).

Listing 3.5 contains an example of a program that contains some operations involving PHP strings and integers. In line 3, a concatenation operation is performed between two strings and it's result is assigned to the variable `$html`. In line 5, there is a sum of two integers whose result is assigned to `$j`.

```
1 <?php
2 $input = $_GET["a"];
3 $html = "Input-" . $input;
4 $i = 10;
5 $j = $i + 1;
6
7 echo $html;
```

Listing 3.5: Example PHP program to demonstrate the simulation of variable operations.

Considering this program, the simulation of variable operations assigns a special marker to `$input` in line 2, to indicate that it contains some input. Next, in line 3, it concatenates

the string `Input-` with the value of variable `$input` to form the simulated value of `$html`. In line 4, the integer `10` is assigned to `$i`. Lastly, in line 5, the value of `$i` is summed with the integer `1` to obtain the simulated value of `$j`. Note that this simulation takes place statically, without ever executing the PHP code. The result of this simulation is a dictionary in which the keys are the names of the variables and the values are their respective simulated values. An example of such dictionary (for the program in Listing 3.5) is shown in Listing 3.6. In this dictionary, the string `::input` corresponds to the special marker inserted in the location where input is located.

```
1 {
2     '$input': '::input',
3     '$html': 'Input-::input',
4     '$i': 10,
5     '$j': 11
6 }
```

Listing 3.6: Result of simulating the variable operations for the program in Listing 3.5.

3.3.2 Pseudocode

In this subsection we will present the main functions used by our approach in the form of pseudocode and describe how they work.

Input: Path to a PHP file; Type of vulnerability
Output: Corrected PHP file

```
1 Function Main (path, vulnerability) :
2     ast ← GenerateAST (path);
3     state ← ProcessPath (ast, vulnerability);
4     corrections ← ProcessCorrections (ast, state, vulnerability);
5     for cor in corrections do
6         InsertLine (path, cor.code, cor.line);
7     end
8 End Function
```

Algorithm 1: Main algorithm of our solution.

Algorithm 1 includes the high level steps of our solution. It starts by generating an Abstract Syntax Tree (AST) that represents the PHP program contained in the input file. Next, it calls the Path Processor to perform the taint analysis and simulate the variable operations. This is done to find the vulnerabilities contained in the code and to compute the state of the program's variables. Next, it calls the Correction Processor to analyze the sensitive sinks, determine the required corrections and where they should be applied. Lastly, it inserts all corrections returned by the Correction Processor in the PHP code to generate the output file.

```
Input: Path to a PHP file
Output: AST representation of the input program
1 Function GenerateAST(path):
2     file ← open (path);
3     code ← file.read ();
4     ast ← Parser.parse (code);
5     close (file);
6     return ast;
7 End Function
```

Algorithm 2: Generation of the AST.

Algorithm 2 is responsible for generating the AST to be used by our solution. It opens the PHP file containing the slice of code and reads its content. Next, it uses a parser to parse the code and produce the corresponding AST. Lastly, it closes the file and returns the generated AST. The generated AST contains a representation of the structure of the program in the form of a tree. In the case of our solution, the AST is a list with no sublists because the input is a slice of code containing a single control flow path. The decision to use ASTs was made because they provide a detailed representation of the original program and there are several tools available to generate them.

Algorithm 3 is the one applied by our Path Processor. It is responsible for performing the taint analysis and simulating the execution of variable operations. This algorithm is composed by several functions. The `ProcessPath` function is responsible for iterating over the AST and calling the remaining functions depending on the type of node it encounters. It must be noted that the fact that our solution's input consists of a slice of code containing a single control flow path makes it easy for the Path Processor to iterate over the AST, because it does not contain any branches. For readability, only the way assignments are treated is shown in this algorithm.

The `ProcessAssignment` function of Algorithm 3 is responsible for dealing with assignments and it first obtains the variable that is assigned to and the assignment's expression. Next, the expression's value is assigned to the variable. If the expression is a binary operator (an operator that has two operands), it's execution is simulated and the result is assigned to the variable. As mentioned in Subsection 3.3.1, this consists of statically simulating the execution of the binary operation (only string concatenations and arithmetic operations are supported) and assigning the result of the simulation to the variable in the state. It is also in this stage that the variable is marked as untainted if the expression is a safe sanitization method for the type of vulnerability being analyzed. Lastly, if the expression is tainted (note that safe sanitization methods are untainted), it's taint status is propagated to the variable. The variable is also marked as untainted if the assignment's expression is a string because this means that the original code contains a string literal being assigned to this variable. A string literal is untainted because there is

```
Input: AST; Type of vulnerability
Output: State of the PHP program
1  Function ProcessPath(ast, vulnerability):
2      state ← {};
3      for node in ast do
4          if node is assignment then
5              ProcessAssignment (node, state, vulnerability);
6          else if [...] then
7              # Remaining types omitted for readability
8      end
9      return state;
10 End Function
11 Function ProcessAssignment(node, state, vulnerability):
12     variable ← node.variable;
13     expression ← node.expression;
14     if expression is a variable or expression is an array access then
15         state [variable] ← expression.value;
16     else if expression is a binary operator then
17         state [variable] ← expression.left.value + expression.right.value;
18     else if expression is a function or expression is a cast then
19         if expression is safe for vulnerability then
20             state [variable].tainted ← false;
21             state [variable].taint_causes ← [];
22         end
23     else if expression is a string then
24         state [variable] ← expression;
25         state [variable].tainted ← false;
26     if expression is tainted then
27         state [variable].tainted ← true;
28         state [variable].taint_causes.append (expression);
29     end
30 End Function
```

Algorithm 3: Algorithm applied by our Path Processor.

no way for a user to influence it.

The `ProcessPath` function also deals with ”combined operators”, such as += and .= (omitted in lines 6 and 7 of Algorithm 3). These operators are a combination of an assignment and a binary operator and they are treated in almost the same way as assignments. The first difference is the fact that the result of simulating their execution is appended to the variable's already simulated value (instead of replacing it). The second difference lies in the taint propagation: even if the operator's expression is untainted, the variable's taint status is not set to untainted. This is done because appending an untainted value to a tainted variable does not make it untainted.

```
   Input: AST; Program state produced by the Path Processor; Type of vulnerability
   Output: List of corrections to be applied
 1 Function ProcessCorrections(ast, state, vulnerability):
 2     corrections ← [];
 3     for node in ast.sensitive_sinks do
 4         for arg in node.arguments do
 5             if arg is a tainted variable then
 6                 if vulnerability is XSS then
 7                     corrections ← CorrectXSSVariable (arg, state,
                         corrections);
 8                 else
 9                     corrections ← CorrectSQLVariable (arg, state,
                         corrections);
10                 end
11             end
12         end
13     end
14     return corrections;
15 End Function
```

Algorithm 4: Algorithm applied by our Correction Processor.

Algorithm 4 represents our solution's Correction Processor. It is responsible for analyzing the sensitive sinks contained in the AST, determining the variables that require correction, what correction they require and the line of code where it should be applied. This algorithm starts by initializing a list that it uses to maintain all corrections that need to be applied. Next, it iterates over all sensitive sinks in the AST and, for each of them, it iterates over it's arguments to find the ones that are tainted, regardless of the type of vulnerability being analyzed. At this stage, if the vulnerability is XSS, the solution's execution flow is directed to Algorithm 5. If the vulnerability is SQLi, the solution's execution flow is directed to Algorithm 6. To conclude, the algorithm returns the list of corrections to be applied.

```
Input: Variable to be corrected; Program state; List of corrections
Output: List with XSS corrections to be applied
1  Function CorrectXSSVariable(arg, state, corrections):
2      if state [arg] contains HTML then
3          for tc in state [arg].taint_causes do
4              if tc not corrected then
5                  if state [tc] contains a script or style tag then
6                      corrections.append (URL-encoding correction for tc in line
                         tc.line);
7                  else
8                      corrections.append (HTML-encoding correction for tc in line
                         tc.line);
9                  end
10                 Mark tc as corrected ;
11             end
12         end
13     else
14         if arg not corrected then
15             if state [arg] contains a script or style tag then
16                 corrections.append (URL-encoding correction for arg in line
                     node.line- 1);
17             else
18                 corrections.append (HTML-encoding correction for arg in line
                     node.line- 1);
19             end
20             Mark arg as corrected ;
21         end
22     end
23     return corrections;
24 End Function
```

Algorithm 5: Correction of XSS variables.

Algorithm 5 is responsible for determining the XSS correction to be applied to a variable and where it should be applied. In this algorithm, the sink's tainted arguments are treated as follows: if the argument contains HTML in it's simulated value, the algorithm obtains it's taint causes (the variable(s) that caused it to become tainted) and adds a XSS correction for the taint cause to the list if it had not been corrected before. If the argument contains no HTML in it's simulated value, a XSS correction for it is added to the corrections list. The XSS corrections to apply consist of a URL-encoding correction if the variable contains a `<script>` or `<style>` tag in it's value and a HTML-encoding correction in the other cases.

```
Input: Variable to be corrected; Program state; List of corrections
Output: List with SQLi correction to be applied to the variable
1 Function CorrectSQLVariable(arg, state, corrections):
2     for tc in state [arg].taint_causes do
3         if tc not corrected then
4             corrections.append (SQLi correction for tc in line tc.line);
5             Mark tc as corrected ;
6         end
7     end
8     return corrections;
9 End Function
```

Algorithm 6: Correction of SQLi variables.

Algorithm 6 is responsible for determining the SQLi correction to be applied to a variable and where it should be applied. In this algorithm, the sink's tainted arguments are treated as follows: the algorithm obtains it's taint causes and, for each of them, adds a SQLi correction to the list if it had not been corrected before. The correction is always applied to the variable's taint causes because SQL queries can not be sanitized in their entirity by a string escaping function. The SQLi correction applied is always the same (a string escaping function).

```
Input: Path to a PHP file; Line of code to insert; Line number to make the insertion
Output: PHP file with the new line inserted
1 Function InsertLine(path, line, number):
2     file ← open (path);
3     lines ← file.readlines ();
4     lines.insertAt (number, line);
5     file.write (lines);
6     close (file);
7 End Function
```

Algorithm 7: Insertion of lines of code in a PHP slice.

Algorithm 7 represents the insertion of lines in a PHP file. First, it opens a file and reads all of it's lines into a list. Next, it inserts the new line of code at a given index in that list. Lastly, it writes all lines to the file and closes it. Only one line of code can be inserted by this algorithm at a time.

Chapter 4

Implementation

This chapter describes the implementation of our solution in the form of a static analysis tool. Firstly, we present the knowledge base for both types of vulnerabilities. Then, we explain the main building blocks and data structures, and we describe the corrections it is capable of applying. To conclude, we justify some decisions that were taken during the implementation.

4.1 Knowledge Base

4.1.1 Entry Points

The tool considers as entry points four of PHP's superglobal arrays: `$_REQUEST`, `$_GET`, `$_POST` and `$_COOKIE`. Any value obtained from one of these arrays is considered to be tainted unless shown otherwise during the taint analysis.

4.1.2 Sensitive Sinks

The sensitive sinks considered for each type of vulnerability are summarized in Table 4.1. Regarding XSS, our tool considers a total of 5 sensitive sinks. Only one of them (the `printf` function) receives a format string as an argument. Apart from the format string, all arguments to these sensitive sinks are treated equally for the purpose of applying corrections. In the case of SQLi, our tool considers a total of six sensitive sinks. Two of them correspond to PostgreSQL and the remaining ones to MySQL. In the case of MySQL, two sensitive sinks use the MySQL extension and the other two use the MySQL

Vulnerability	Sensitive Sinks
SQLi	`mysql_query`, `mysql_unbuffered_query`, `mysqli_query`, `mysqli_real_query`, `pg_query`, `pg_send_query`
XSS	`echo`, `print`, `die`, `exit`, `printf`

Table 4.1: Sensitive sinks considered by our tool.

Type of Sanitization Method		Method/Function
Generic	Function	`intval`, `floatval`, `strlen`, `strpos`, `md5`, `sha1`, `base64_encode`, `password_hash`, `count`, `round`, `bin2hex`
	Cast	Casts to `int`, `integer`, `float`, `double`, `real`, `bool`, `boolean`
XSS	Function	`htmlentities`, `htmlspecialchars`, `urlencode`, `rawurlencode`, `http_build_query`
SQLi	Function	`mysql_real_escape_string`, `mysqli_real_escape_string`, `mysql_escape_string`, `mysqli_escape_string`, `pg_escape_string`, `pg_escape_literal`, `pg_escape_bytea`
Filter	Generic	`FILTER_SANITIZE_NUMBER_FLOAT`, `FILTER_SANITIZE_NUMBER_INT`
	XSS	`FILTER_SANITIZE_ENCODED`, `FILTER_SANITIZE_SPECIAL_CHARS`, `FILTER_SANITIZE_FULL_SPECIAL_CHARS`

Table 4.2: Sanitization methods supported by our tool.

Improved extension.

4.1.3 Sanitization Methods

The sanitization methods supported by the tool are shown in Table 4.2. Apart from the casts and filters, all sanitization methods are function calls. In the case of the generic sanitization functions, the tool considers safe the `intval` and `floatval` functions as explained in Subsection 2.2.1. In addition to these functions, it also considers safe nine other functions that were added to this list during the implementation. These functions are not usually seen as sanitization functions but they perform that role in practice because they can not return data that will lead to an attack.

Firstly, we added the `strlen` and `strpos` functions because they return the length of a string and the index of a substring within a string, respectively. This means that they always return integers. Secondly, we added the `md5` and `sha1` functions because they return cryptographic hashes that are composed solely of hexadecimal digits. This means that they can not return malicious Javascript or SQL (with very high probability). The `base64_encode` function is also in this list because it returns a base64-encoded version of the string it receives as argument, meaning that it also can not return malicious data. Next, we added the `password_hash` and `bin2hex` functions because they return a

hashed version of a password and an hexadecimal representation of a string, respectively. Both of these returned values can not contain malicious data. Lastly, we added the `count` and `round` functions because they always return numbers. The former counts the number of elements in an array and the latter rounds a float to a given precision.

Regarding the casts, our tool considers as safe casts to numeric types (`int`, `integer`, `float`, `double` and `real`) and casts to booleans (`bool` and `boolean`). The casts to numeric types were mentioned in Subsection 2.2.1 and the casts to booleans will convert any value to `true` or `false`.

The sanitization functions that the tool regards as safe for XSS are the ones mentioned in Subsection 2.2.2. For SQLi, the tool views as safe functions of the `*_escape_string` family for MySQL and PostgreSQL. This family of functions was described in Subsection 2.2.3. In addition to these functions it also considers as safe the `pg_escape_literal` and `pg_escape_bytea` functions. The former escapes a string for insertion into a text field and the latter escapes a string for insertion into a field of type `bytea` in PostgreSQL.

Lastly, the tool sees two generic filters as safe for both types of vulnerabilities. They are `FILTER_SANITIZE_NUMBER_FLOAT`, that performs in the same way as `floatval`, and `FILTER_SANITIZE_NUMBER_INT` that executes similarly to `intval`. It also assumes as safe three filters specific to XSS vulnerabilities. The first one carries out URL-encoding of a string in the same way as calling `urlencode` and is named `FILTER_SANITIZE_ENCODED`. The other two (named `FILTER_SANITIZE_SPECIAL_CHARS` and `FILTER_SANITIZE_FULL_SPECIAL_CHARS`) do HTML-encoding in the same manner as calling `htmlspecialchars` with the `ENT_QUOTES` flag. For these filters to be considered secure, they must be used in a call to `filter_var`.

4.2 The PHPly Parser

PHPly[1] is a parser written in Python for the PHP language. It takes PHP code as a string and returns an AST in the form of a Python list. This list contains nodes built by PHPly itself, which contain the line number where the node is in the original code and include sublists when necessary for the representation of code blocks (for example, the body of a loop). PHPly supports both PHP 5 and PHP 7, the two most used versions of PHP nowadays. Listings 4.1 and 4.2 provide an example of a PHP program and the respective AST. PHPly was chosen as the parser to be used in our tool due to it's simplicity, ease of installation and the fact that it supports PHP 7.

In Listing 4.2, line 2 of the AST contains a node of type `Assignment` and corresponds to the assignment of the value 1 to variable `$a` made on line 2 of the original program (in Listing 4.1). Lines 3 and 4 of the AST correspond to the assignment made to `$b` on line 3 of the program. The + operation is represented in the AST by a node of

[1] https://github.com/viraptor/phply

```
1 <?php
2 $a = 1;
3 $b = $a + 1;
4 echo $b;
```

Listing 4.1: Example PHP program to demonstrate PHPly.

```
1 [
2     Assignment(Variable('$a'), 1, False),
3     Assignment(Variable('$b'),
4                BinaryOp('+', Variable('$a'), 1), False),
5     Echo([Variable('$b')])
6 ]
```

Listing 4.2: Example AST generated by PHPly for the program in Listing 4.1.

type `BinaryOp`. Lastly, line 5 of the AST corresponds to the call to `echo` made in line 4 of the original program, with variable `$b` as an argument. The line numbers where each node is located in the original program are not shown in the AST's textual representation but are stored internally.

4.3 Data Structures

This section presents the main data structures used by our tool and justifies the need for them. It must be noted that, due to the fact that a slice of code contains a single control flow path, it's representation in the form of an AST generated by PHPly is composed by a single list, with no sublists. This is important because it influenced the design of some of the data structures we created.

4.3.1 Variable Definition

`VariableDefinition` is a Python class created by us to maintain information about program variables during the analysis. One instance of this class is created while processing the program to represent each definition of a PHP variable. Every variable is characterized by the following attributes: the variable's name, simulated value (according to the idea described in Subsection 3.3.1) and taint status (whether it is tainted or untainted). The class maintains these attributes and, in addition, it also keeps the name(s) of the variable(s) that caused the represented one to become tainted (referred to as taint causes), along with the line number(s) where that happened. Lastly, it includes a definition counter that keeps track of the number of times the variable was redefined in the PHP program. This was added during the implementation to allow the tool to better deal with variables that receive input from multiple entry points along the execution of the program.

The need for a counter is illustrated in Listing 4.3. In this program, the tool creates a `VariableDefinition` object to represent the definition of `$a` on line 3. Because this is the first time `$a` is defined, the counter is set to zero. In line 5, another `VariableDefinition` object is created to represent `$b`. Since `$b` contains the value of `$a`, it is marked as tainted and a taint cause is added to `$b`. This taint cause consists of the variable `$a` and respective counter (at this point it is zero). Next, on line 7, `$a` is redefined and a new `VariableDefinition` object is created to represent it. This new object contains the counter set to one because this is the second time that `$a` was defined in the program. On line 9, a new variable `$c` is created and, because it is tainted, a taint cause is added to it. This new taint cause is the same as the one added to `$b`, except that the counter is now set to one. On lines 11 and 12, the tool has to decide what corrections to apply and where to apply them. The tool will correct `$b` first and it will decide to correct `$a` (with `$a`'s counter at zero) on line 4 because that is `$b`'s taint cause. Next, the tool will correct `$c` and it will decide to correct `$a` (with `$a`'s counter at one) on line 8 because that is `$c`'s taint cause. Note that `$b` and `$c` can not be corrected themselves because they contain developer-provided HTML.

```
1  <?php
2
3  $a = $_GET["a"];
4
5  $b = "<p>" . $a . "</p>";
6
7  $a = $_GET["b"];
8
9  $c = "<p>" . $a . "</p>";
10
11 echo $b;
12 echo $c;
```

Listing 4.3: Example of a program to illustrate the need for a definition counter.

It is at this point that the need for a counter becomes clear: the counter allows the tool to know that `$b` and `$c`'s taint causes are two "different versions" of `$a`. This means that the tool will correct `$a` twice (in lines 4 and 8) and make the program safe, without correcting the same definition of `$a` more than once. Without the counter, the tool would see both taint causes as being the "same version" of `$a` and, because it contains mechanisms to prevent it from correcting the same variable more than once, it would correct `$a` only in line 4, leaving the program vulnerable due to the input `$a` receives on line 7.

It is important to note that the counter does not tell the tool where the correction has to be inserted. It just allows the tool to know that a given variable may require more than one correction. The locations where the corrections have to be inserted are determined based on the information contained in the AST.

4.3.2 Program State

The `ProgramState` is produced by the Path Processor and it contains the state of all program variables known by the tool at a given point in the analysis. It is composed of a dictionary in which the keys are strings with the name of the variables and the values are lists of `VariableDefinition` objects, defined by the attributes mentioned in Subsection 4.3.1.

This data structure is the most important one in our tool. It is mainly used by the Path Processor and the Correction Processor to check whether a variable is tainted or untainted. It is also used by both of these components to obtain a variable's simulated value. Considering a variable named `V`, the Path Processor uses the `ProgramState` to obtain `V`'s simulated value for the purpose of using it in the simulation of an operation involving `V`. Considering the same variable, the Correction Processor uses it's simulated value to determine what correction to apply and where.

The reason why the values in this dictionary are lists is because the lists keep all of the variable's definitions. As shown in Subsection 4.3.1, keeping track of all definitions of a variable allows the tool to better correct files in which the same variable receives input from multiple entry points at different stages in the program. The fact that this data structure contains lists allows the tool to know the value to use in the definition counter of a new `VariableDefinition` object (it uses the length of the list). It also allows the tool to obtain the simulated value and taint status of any of a variable's definitions.

4.3.3 Program State by Sensitive Sink

The `ProgramStateBySensitiveSink` is a data structure (produced by the Path Processor) used to maintain the state of the program when the control flow reaches each one of the sensitive sinks it might contain. This data structure consists of a dictionary in which the keys are integers and the values are `ProgramState` dictionaries following the structure described in Subsection 4.3.2. The indexes of the sensitive sinks in the AST (note that the AST is a list) are used as keys to identify them in this dictionary. The values are `ProgramState` dictionaries that represent the state of the program's variables when it's control flow reaches the corresponding sensitive sink.

This data structure is necessary because our Path and Correction Processors perform their tasks at different stages of the tool's execution. This means that, when the Correction Processor is started, the taint analysis and simulation of variable operations are already completed. If the Correction Processor had to perform it's task based solely on the `ProgramState`, it would only have access to the `ProgramState` representing the end of the program. This problem sometimes caused the Correction Processor to fail to correct programs in which the sensitive sink(s) appeared closer to the beginning.

An illustrative example of this situation is provided in Listing 4.4. In this example,

there is a sensitive sink located on line 6 and another one on line 9. When the program's control flow reaches line 6, the variable `$a` is tainted, making the program vulnerable. However, in line 8, `$a` is assigned the value `0`, thus becoming untainted. This means that the second sensitive sink in line 9 has no influence on the security of the program.

```
1 <?php
2
3 $tainted = $_GET["a"];
4
5 $a = "Input: " . $tainted; // $a is tainted
6 echo $a;
7
8 $a = 0; // $a is untainted at the end of the program
9 echo $a;
```

Listing 4.4: Illustrative example for the data structure described in Subsection 4.3.3.

Considering the same example, when the Path Processor finishes the taint analysis, variable `$a` is marked as untainted because that is it's state at the end of the program. Without the data structure described in this subsection, the Correction Processor would only have access to the program's final state. This would cause it to see variable `$a` as untainted and fail to correct it for the `echo` in line 6. Thanks to this data structure, the Correction Processor can know what is the state of the program when it reaches the `echo` in line 6 and also when it reaches the `echo` in line 9. This means that the Correction Processor knows that `$a` is tainted in line 6 and untainted in line 9, thus applying the necessary correction to the first sensitive sink.

4.3.4 Lines of Code with HTML

The information about what lines of code contain HTML is kept in a dictionary where the keys are the line numbers and the values are true if the line contains HTML or false otherwise. The dictionary is built gradually as the simulation of the variable operations is taking place. When the Path Processor encounters HTML in the path, it marks all lines that the HTML spans over as true in the dictionary. This means that, along the tool's execution, the fact that a given line number `N` is in this dictionary is enough to know that `N` contains HTML. This allows the dictionary to take up less space in memory because it will only contain entries for lines of code with HTML.

The need for this data structure is motivated by the fact that sometimes the tool has to insert corrections in lines of code that contain HTML. This means that the line(s) of code to be inserted must be surrounded by PHP tags or the PHP interpreter will treat them as HTML and display them on the output, instead of executing them. With the help of this data structure, the tool can check whether a given line of code contains HTML and, if it does, it surrounds the correction with PHP tags, to ensure it is executed. This is illustrated by the code in Listing 4.5. Following the design described in Subsection 3.1.1, the tool

would insert the correction immediately before line 4 (line 3). In order for the correction to have an effect in this case, it must be surrounded by PHP tags. This data structure allows the tool to know that lines 2 and 3 contain HTML and any correction inserted in them must be surrounded by PHP tags. Listing 4.6 shows an example of a corrected program in which the correction required PHP tags.

```
1 <?php $tainted = $_GET["a"]; ?>
2 <html>
3 <body>
4 <?php echo $tainted; ?>
5 </body>
6 </html>
```

Listing 4.5: Illustrative example with PHP and HTML in the same program.

```
1 <?php $tainted = $_GET["a"]; ?>
2 <html>
3 <body>
4 <?php $tainted = htmlentities($tainted, ENT_QUOTES); ?>
5 <?php echo $tainted; ?>
6 </body>
7 </html>
```

Listing 4.6: Corrected version of the program in Listing 4.5 (line 4 contains the correction).

4.3.5 Corrected Variables

This data structure is used by the Correction Processor to keep the names (and respective definition counters) of the variables that have been corrected by the tool, preventing the tool from correcting the same variable more than once. It consists of a dictionary in which the keys are strings in the format `name:counter`, where `name` is the name of the variable and `counter` is the value of it's definition counter. The counter is included in the key to distinguish between multiple definitions of the same variable. The values in this dictionary are true if a correction has been applied to the variable or false otherwise. As with the dictionary described in Subsection 4.3.4, this dictionary is built gradually as the corrections are applied. This means that it only contains entries for variables that have been corrected and that the tool just has to check whether a key exists in it to know if a variable has been corrected.

4.4 Corrections Applied

To any given variable that requires correction, the tool applies one of four possible corrections. The tool is capable of applying three corrections for XSS and one for SQLi. Regard-

ing SQLi, the tool always applies a call to a MySQL (or MySQLi) string escaping function. The used function is `mysqli_real_escape_string` when the sensitive sink has a database connection as it's first parameter and `mysql_real_escape_string` in any other case. Currently, the only sensitive sinks considered that have a connection as their first parameter are: `mysqli_query`, `mysqli_real_query` and `pg_send_query`. When a connection is available, the correction consists of the following code: `$t = mysqli_real_escape_string($c, $t);`, where `$c` is the name of the connection and `$t` is the name of a tainted variable (it may also be an access to an array). When a connection is not available, the following correction is inserted instead: `$t = mysql_real_escape_string($t);`, where `$t` is once again the name of a tainted variable.

For XSS, the tool has three corrections at it's disposal: a HTML-encoding correction, an URL-encoding correction and a special correction for format strings. The format string correction is more complex and will be described in detail in Subsection 4.4.1. The HTML-encoding correction consists of a call to the `htmlentities` function with a tainted variable as it's first argument and the `ENT_QUOTES` flag as it's second argument. This correction is applied in all situations when the other corrections can not be applied and it consists of adding the following code: `$t = htmlentities($t, ENT_QUOTES);`, where `$t` is the name of a tainted variable (as with SQLi, this can be an access to an array). `htmlentities` was chosen as the HTML-encoding function to be used in this type of correction because it is widely recommended by the community. In the case of the URL-encoding correction, the tool instead adds the following code: `$t = rawurlencode($t);`, where `$t` has the same meaning as before. The URL-encoding correction is applied when the variable is being included inside of a `<script>` or `<style>` tag. `rawurlencode` was chosen over `urlencode` as the URL-encoding function to be used in this type of correction because it also encodes the + character, which is a valid operator in Javascript.

4.4.1 Format String Correction

The format string correction is applied to tainted variables that are part of a format string in a call to the `printf` sensitive sink or a call to `sprintf` that is an argument to another XSS sensitive sink. Both of these functions expect to receive a format string (similar to the format strings used in the C language) as their first argument. The format string can not be corrected using the two previously described XSS corrections because they would result in the encoding of some of the format specifiers, thus breaking the application's output. For this reason, we developed a way of correcting format strings without breaking their original output. This is the only correction applied by our tool that involves the addition of more than one line of code. Listing 4.7 presents an example of the correction applied to format strings. The correction itself starts on line 6 of the listing and ends on

line 20, inclusive.

```
1  <?php
2
3  $format = $_GET["f"];
4  $input = $_GET["i"];
5
6  $matches = array();
7  preg_match_all("/(?<=%')./", $format, $matches);
8
9  $format = preg_replace("/(?<=%')./", "#", $format);
10 $format = htmlentities($format);
11 $format = preg_replace("/(?<!%)'/", "&#039;", $format);
12
13 $matchIdx = 0;
14 for ($ptr = 2; $ptr < strlen($format); $ptr++) {
15     if ($format[$ptr] == "#" && $format[$ptr - 1] == "'" &&
16             $format[$ptr - 2] == "%") {
17         $format[$ptr] = $matches[0][$matchIdx];
18         $matchIdx++;
19     }
20 }
21
22 printf($format, $input);
```

Listing 4.7: Example of the correction applied to a format string.

In PHP, out of all characters that can form a format specifier, only the single quote is converted by a call to `htmlentities` (if the `ENT_QUOTES` flag is in use). The single quote is used in a format string to specify a character to be used as padding for the argument. The padding character is the one immediately to the right of the single quote. For example, the format specifier `%'09s` will pad a string with zeroes on the left until it is 9 characters in length. If the original string is 9 or more characters in length, no padding will be added. Because the padding is part of the application's output, it is important that the structure of these format specifiers is not modified by a correction.

Taking this into consideration, our correction does the following: In lines 6 and 7, it locates all padding characters and saves them to the `$matches` array. In line 9, all padding characters are replaced by the # character, to prevent them from being modified next. In line 10, a call to `htmlentities` is made without any flag, to prevent it from modifying single quotes (all other applicable characters are still converted). In line 11, any single quote that is not part of a format specifier (not preceeded by `%`) is replaced by it's equivalent HTML-encoded representation. Lastly, in lines 13 to 20, the padding characters that were saved to the `$matches` array are put back in their place to be part of the output. This process HTML-encodes any applicable character while maintaining the original paddings.

Before concluding the description of this correction, there are some things than must

be noted: Firstly, this correction uses a HTML-encoding function and thus inherits all limitations of this type of functions. Secondly, the use of URL-encoding functions is not possible in this situation because they would encode many more characters (including the ones that form the format specifiers) than HTML-encoding ones. Lastly, in order to prevent the names of the variables created by this correction from interfering with others that already exist in the program, our tool appends a random number to the end of the name of each variable created by this correction. As an example, the `$matches` array will be named `$matches_NNNN` in a real correction, where `NNNN` is a random number (greater than zero) generated by the tool. Despite the use of this technique, it is still possible for the added variables to have the same name as ones that already exist in the program. We believe this is not a problem because the correction for format strings is rarely applied and, when it is applied, the use of random numbers makes the chances of interfering with existing variables extremely low.

4.5 Implementation Decisions

In this section, we will describe several decisions that were taken during the implementation of our tool to deal with some problems that arose during it's development.

If statements

We added limited support for if statements during the implementation, despite the fact that our definitition of a slice of code does not include them. When the tool encounters an if statement in the path, it processes only one of it's branches as if it was part of the remaining path. To achieve this, the tool analyzes all branches of the if statement and keeps the first one to contain a sensitive sink in it or the last of them if none contain sensitive sinks. We do not believe this to cause problems in our tool because it's input consists of slices of code that do not contain if statements. However, we added this capability to allow the tool to deal with slightly more complex code than a simple slice.

Detection of HTML Tags

We also decided to use regular expressions to detect if a string contains HTML tags in it's simulated value. This has some limitations when dealing with malformed HTML tags but it rarely detects HTML as not being HTML, which minimizes the impact of this limitation on our tool's operation. In the cases when non-HTML is detected as HTML, the tool will still apply a valid correction. It will just be applied in a different line of code. Regular expressions were chosen for this purpose because they are supported natively by Python and produced satisfying results.

Also, due to the fact that PHPly only includes in the AST the line number where a block of HTML starts, we decided to count the number of line breaks contained in the

HTML itself in order to know the line number where it ends and allow the tool to build the dictionary described in Subsection 4.3.4.

To decide between the URL-encoding or HTML-encoding to correct tainted variables (that are not part of a format string) in the case of XSS, the tool needs to know whether the sensitive sink is inside of a `<script>` or `<style>` tag. To achieve this, when the tool encounters a HTML block in the path, it counts the number of opening and closing `<script>` and `<style>` tags. The number of closing tags is then subtracted from the number of opening tags. If the result is greater than zero when a sensitive sink is encountered, this means that the sink is inside of one of these tags, thus requiring a URL-encoding correction. This count is done with regular expressions. The tool is also capable of detecting cases when the sensitive sink contains one of these two tags as part of it's argument (e. g. `echo "<script>" . $_GET["a"] . "</script>";`) and apply a URL-encoding correction. In the other cases, a HTML-encoding correction is applied.

Undefined Variables and Functions

During the taint analysis, if the tool encounters a variable that is not defined, it means that variable is not defined in the original program. Because the tool has no way of knowing whether the variable is tainted or untainted, it considers it as untainted for the purpose of the taint propagation. This was done to prevent the taint analysis from producing large numbers of tainted variables that could lead to many false positives. For the purpose of simulating the variable operations, an unknown variable is considered to have an empty string as it's value. Regarding function calls, any function that is not a sanitization function for the vulnerability being analyzed is considered to return a tainted value if the function contains at least one tainted variable as an argument. This was done to prevent the tool from producing false negatives for vulnerabilities that arise from the use of sanitization functions that are unsafe in most contexts. For the purpose of simulating the variable operations, the value of the first parameter is used if the function is safe or the value of the first tainted parameter is used if the function is not safe.

Correction of Arrays

In order for the tool to apply a correction to a variable, it needs to know the name of that variable in order to include it in the line(s) of code to be added. This is trivial to do when the correction is applied to an actual variable because PHPly includes the name of the variable in the AST. It is harder to do when the correction has to be applied to an array because only the name of the array itself (e. g. `$_GET`) is included in the AST. The PHP code that provides the key to access the array is included in the form of a "smaller" AST inside the program's AST. Due to the fact that reversing an AST is very difficult, we decided that the tool would need to read the array's key from the original PHP file.

To do this, the tool first reads from the file the line of code where the access to the array occurs (the line number is part of the AST). Given that line of code in the form of a string, the tool locates the array's name in it and reads the PHP code located between the square brackets that limit the array's key. Situations where the key to access an array includes an access to another array are supported by the tool. All keys used to access arrays are read from the original PHP file, regardless of how complex they are. As an example, considering `$_GET["user"]` as the array access to be corrected, the tool will extract `$_GET` from PHPly's AST and concatenate it with the string `["user"]` read from the original file. It is important to note that this technique suffers from a limitation: if the array access spans over multiple lines of code in the original file, this process will fail. We believe this to have a small impact on our tool due to the fact that it's input consists of simplified programs.

Chapter 5

Evaluation

This chapter describes the evaluation performed of our tool. To evaluate the proposed solution, we first used the developed tool to analyze and correct some automatically generated test cases taken from the Software Assurance Reference Dataset (SARD). Secondly, we took six vulnerable web applications written in PHP from Exploit-DB[1] to test our tool's ability to process real code. We focused most of our evaluation on XSS due to time restrictions.

5.1 Software Assurance Reference Dataset (SARD)

SARD[2] is a dataset created and maintained by the National Institute of Standards and Technology (NIST). It contains several thousand test cases that were generated automatically, which together contain different types of vulnerabilities in different programming languages. Each test case contains only one vulnerability corresponding to a given CWE[3] (Common Weakness Enumeration) category in one programming language. Since our work is focused on PHP, we are only interested in test cases for that programming language. Stivalet and Fong describe the generation process in [23]. Each PHP test case consists of a single file that contains three sections:

Input: This is the section of the program that obtains input from the user. SARD contains test cases with distinct methods of getting input of varying complexity, ranging from obtaining the value of a superglobal array directly to obtaining it via a class method call. All test cases include this section.

Sanitization: This is the section of the program that sanitizes the data. This section is optional as some of the test cases have no sanitization. The sanitization methods vary from the use of a ternary condition to the use of casts to numeric types. It is

[1] https://www.exploit-db.com/
[2] https://samate.nist.gov/SRD/
[3] https://cwe.mitre.org/

```
1  <!DOCTYPE html>
2  <html>
3  <head/>
4  <body>
5  <div>
6  <?php
7  $tainted = $_POST['UserData'];
8
9  $tainted = htmlspecialchars($tainted, ENT_QUOTES);
10
11
12 echo $tainted ;
13 ?>
14 </div>
15 <h1>Hello World!</h1>
16 </body>
17 </html>
```

Listing 5.1: Test case 192719 from SARD.

important to note that some test cases contain unsafe sanitization methods, such as the use of unsafe functions.

Sensitive Sink: This is the section where the potentially unsafe input reaches a sensitive sink. Like the first section, all test cases contain this section.

Listing 5.1 provides an example of test case 192719 from SARD, where line 7 corresponds to the Input section, line 9 corresponds to the Sanitization section and line 12 contains the Sensitive Sink. The file names of the SARD test cases for PHP have the following structure:

`CWE_<CWE ID>__<input source>__<sanitization method>__<location where input is inserted>.php`

The large number of test cases combined with the variety of sanitization methods makes the SARD an ideal source of test cases for evaluating a tool such as the one we developed. It allowed us to test the tool's capability to reason about how secure a sanitization method is.

5.1.1 XSS Dataset Summary

We gathered a total of 1764 XSS test cases from SARD. All of these test cases contain a single control flow path, meaning that they fit our definition of a slice of code perfectly.

While analyzing some of the test cases manually, we discovered that some of SARD's test cases are mislabelled. There are safe test cases that are considered as unsafe and vice-versa. For this reason, we ran a more thorough analysis to determine the actual label that

	SARD Label	Our Label
Safe	1056	1344
Unsafe	708	420

Table 5.1: Number of safe and unsafe XSS test cases according to the two labels.

the test cases should have. Table 5.1 provides the total number of safe and unsafe test cases according to the two labellings in consideration, the one provided by SARD and our own. Our dataset contains 1396 test cases tagged equally and 368 test cases tagged differently. Out of the test cases labelled equally, 1016 are safe and 380 are unsafe. Among the test cases that were tagged differently, there are 328 cases considered ”unsafe” by SARD and ”safe” by us, plus 40 cases labelled ”safe” by SARD and ”unsafe” by us. The reasons for this large discrepancy will be explained in detail in Subsection 5.1.7.

Our labels were generated by using small automation scripts to iterate over the test cases and write their new label to a CSV file. Due to the fact that the SARD test cases contain well-structured file names, the name of a test case file contains all the information needed to determine whether it is safe or unsafe. We trivially considered all test cases with no sanitization unsafe. We also considered all test cases that use casts or the ternary operator as safe, due to the fact that no malicious data can reach the sensitive sink. Regarding the test cases that use function calls, we considered safe all test cases that use functions marked as ”Always Safe” in Table 5.2. For the remaining test cases, we analyzed them manually in more detail paying special attention to the sanitization function used and the location where the input is included in the program’s output. When we were unsure about what label to assign to a given test case, we tried to inject Javascript into it’s output by providing it with malicious input. When the injected Javascript was executed, we labelled the test case as ”unsafe”. Otherwise, we associated ”safe” to the test case.

5.1.2 SQLi Dataset Summary

We gathered a total of 100 SQLi test cases from SARD. All of these test cases contain a while loop to iterate over the query’s result. However, we considered them to fit our definition of a slice of code because this loop occurs after the sensitive sink, which means that it has no influence on the tool’s capability to find vulnerabilities.

While analyzing the test cases manually, we discovered that there are two safe test cases that are labelled as unsafe by SARD. The remaining test cases are labelled correctly. Table 5.3 shows the total number of safe and unsafe test cases according to the two labels. Due to the smaller size of the dataset for SQLi, our labels were generated manually by analyzing all test cases. The two mislabelled test cases use a PHP filter equivalent to call the function `htmlspecialchars` with the `ENT_QUOTES` flag set. Although this is usually considered to be unsafe for SQLi, it is safe in these two test cases. One of the

Function	Always Safe for XSS in SARD	Observations
intval	Yes	Always returns an integer
floatval	Yes	Always returns a float
preg_replace	No	It's safety depends on the regular expression used
addslashes	No	Only sanitizes quotes and slashes
htmlentities	No	Is unsafe if it's result is included in a `<script>` tag, `<style>` tag or a Javascript event handler
htmlspecialchars	No	
filter_var	No	It's safety depends on the filter used
mysql_real_escape_string	No	Is inadequate to prevent XSS and performs similarly to `addslashes`.
http_build_query	Yes	URL-encodes any non-alphanumeric characters, preventing the writing of Javascript
rawurlencode	Yes	
urlencode	Yes	

Table 5.2: Safety of the XSS sanitization functions in our dataset.

	SARD Label	Our Label
Safe	85	87
Unsafe	15	13

Table 5.3: Number of safe and unsafe SQLi test cases according to the two labels.

mislabelled test cases is shown in Listing 5.2.

5.1.3 Sources of Input

Overall, the test cases for both types of vulnerabilities contain input from four different sources:

`$_GET` **directly**: The test cases with this input source assign a value from the `$_GET` array directly to a variable called `$tainted`. This variable is then used by the program.

`$_POST` **directly**: The test cases with this input source assign a value from the `$_POST` array directly to a variable called `$tainted`. This variable is then used by the program.

`$_GET` **via array**: In these test cases, a value from the `$_GET` array is assigned to index 1 of another array. The value of index 1 of this new array is then assigned to a variable named `$tainted`, that is later used by the program.

`$_POST` **via unserialize**: In these test cases, a serialize string is received from the `$_POST` array. This string is then passed to `unserialize` and the result of this

```php
<?php

$tainted = $_POST['UserData'];

$sanitized = filter_var($tainted, FILTER_SANITIZE_SPECIAL_CHARS)
    ;
  $tainted = $sanitized ;


$query = "SELECT * FROM '". $tainted . "'";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
    '); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Listing 5.2: Test case 166331, marked as ”unsafe” by SARD and ”safe” by us.

function call is assigned to a variable called `$tainted`. As in the previous input sources, this variable is then used by the program.

5.1.4 XSS Sanitization Methods

Our XSS test cases contain four major groups of sanitization methods, as described next.

No sanitization: contain no form of sanitization, making all test cases in this category unsafe.

Function call: use a single function call to perform the sanitization. There are examples with distinct functions, but they all have one thing in common: they receive a variable as one of their arguments and return a sanitized version of that variable. It is important to note that not all function calls are secure. This is the only category that contains both safe and unsafe test cases, depending on the function used and the location where the value is included on the resulting web page. The safety of the functions in our XSS dataset is shown in greater detail in Table 5.2. Note that even the functions not marked "Always Safe" can be safe in certain conditions.

Cast to a numeric type: the value of the tainted variable is cast to a numeric type. The cast can be done explicitly via the cast operation (e.g., `(int) $v`) or implicitly via an arithmetic operation. In the second case, the numeric value 0 is summed to the tainted variable (e.g., `$t = $t + 0;`), making PHP treat the variable as a number regardless of it's original type. Some of the test cases perform the sum using the += operator while others use the longer syntax. All test cases in this category are safe because the value that reaches the output is always a number, thus preventing the attacks.

Ternary condition: use the ternary operator to test the input and assign to it one of two allowed values, based on the result of the test. This effectively implements a white list. All test cases in this category are safe because only one of two safe values can reach the output.

All test cases contain a variable with the name `$tainted` that contains the program's input. These four types of sanitization methods provide a good variety of behaviors to test our tool's taint propagation mechanism. Table 5.4 summarizes the number of test cases for the sources of input and sanitization methods contained in our XSS dataset.

5.1.5 SQLi Sanitization Methods

Our SQLi test cases contain four major groups of sanitization methods, as described next.

No sanitization: contain no form of sanitization, thus making all test cases in this category unsafe.

			Input Source				Total
			$_GET directly	**$_POST directly**	**$_GET via array**	**$_POST via unserialize**	
Sanitization Method	**No Sanitization**		21	21	21	21	84
	Function Call	intval	21	21	21	21	84
		floatval	21	21	21	21	84
		preg_replace	42	42	42	42	168
		addslashes	21	21	21	21	84
		htmlentities	21	21	21	21	84
		htmlspecialchars	21	21	21	21	84
		http_build_query	21	21	21	21	84
		mysql_real_ escape_string	21	21	21	21	84
		filter_var	63	63	63	63	252
		rawurlencode	21	21	21	21	84
		urlencode	21	21	21	21	84
	Cast to a Numeric Type	Cast into int	21	21	21	21	84
		Cast into float	21	21	21	21	84
		Cast via += 0	21	21	21	21	84
		Cast via + 0	21	21	21	21	84
		Cast via += 0.0	21	21	21	21	84
	Ternary Condition		21	21	21	21	84
Total			441	441	441	441	1764

Table 5.4: Summary of the XSS test cases we collected from SARD, their sources of input and sanitization methods.

Function call: use a single function call to perform the sanitization. There are examples with distinct functions, but they all have one thing in common: they receive a variable as one of their arguments and return a sanitized version of that variable. It is important to note that not all function calls are secure. This is the only category that contains both safe and unsafe test cases, depending on the function used and the location where the value is included in the resulting query. The functions used in the SQLi dataset are the same ones used for XSS, with the exception of the URL-encoding ones. Only the `mysql_real_escape_string` function has some unsafe test cases associated with it.

Cast to a numeric type: the value of the tainted variable is cast to a numeric type, as in the case of XSS. The casts can also be done via the use of a numeric format specifier (such as `%u`) in a call to `sprintf` to convert the input into a numeric type. All test cases in this category are safe because the value that reaches the query is always a number, thus preventing attacks.

Ternary condition: as with XSS, the ternary operator is used to test the input and assign to it one of two allowed values, based on the result of the test. All test cases in this category are safe because only one of two safe values can reach the query.

5.1.6 Sensitive Sinks

Regarding the XSS sensitive sinks, all test cases use PHP's `echo` statement. Some test cases simply provide the `$tainted` variable as an argument to `echo` while others contain a concatenation of `$tainted` with other strings. The location where the input is included in the HTML also varies, ranging from the content of a `<div>` tag to the content of a Javascript event handler.

As for the SQLi sensitive sinks, all test cases use the `mysql_query` function. All test cases provide a variable named `$query` as an argument to the sensitive sink. The location where the input is included in the query varies, ranging from the `WHERE` clause to the name of a table.

These factors contribute to enhance the diversity of the test cases for our tool because the safety of some sanitization functions can change depending on the location where the input is included in a query or the HTML.

5.1.7 Explanation for Mislabelling in the XSS Dataset

As explained in Subsection 5.1.1, there is a large discrepancy between SARD's labels and ours in the XSS dataset. We performed a more detailed analysis of the test cases that were labelled differently to determine the root causes for this discrepancy and found the following reasons, and provide an example of each:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <script>
5  <?php
6  $tainted = $_POST['UserData'];
7
8  $tainted = urlencode($tainted);
9
10 //flaw
11 echo $tainted ;
12 ?>
13 </script>
14 </head>
15 <body onload="xss()">
16 <h1>Hello World!</h1>
17 </body>
18 </html>
```

Listing 5.3: Test case 192860, marked as ”unsafe” by SARD and ”safe” by us.

URL-encoding: SARD contains some test cases marked as ”unsafe” that use sanitization functions to perform URL-encoding (line 8 in Listing 5.3). These functions encode any non-alphanumeric characters, thus preventing the writing of Javascript, leading us to tag them as ”safe”.

Quote escaping: Some test cases use sanitization methods that escape quotes by preceding them with backslashes (line 8 in Listing 5.4) and later include their input inside of an HTML quoted attribute. Although SARD lists them as ”safe”, we marked them as ”unsafe” because we were able to execute Javascript by providing a malicious input. The input `1" onmouseover=alert(1) a="` can trigger the vulnerability by closing the `id` attribute of the `<div>` tag contained on line 11 and introducing the `onmouseover` attribute with malicious Javascript. Note that most browsers will accept a HTML attribute that contains backslashes as part of it’s value.

Ternary condition: Some SARD test cases that use the ternary operator (line 9 in Listing 5.5) are marked as ”unsafe”. We marked them as ”safe” because no malicious input can reach the sensitive sink. The ternary operator used in these test cases assigns the string `'safe1'` to the variable `$tainted` if it’s value is equal to `'safe1'` or the string `'safe2'` in any other case. This means that regardless of the value originally contained in `$tainted`, the ternary operator will always assign to it one of two safe values. As an example, consider that the variable `$tainted` contains the value `<script>`. In this case, the ternary operator’s condition will evaluate to

```
1  <!DOCTYPE html>
2  <html>
3  <head/>
4  <body>
5  <?php
6  $tainted = $_GET['UserData'];
7
8  $tainted = addslashes($tainted);
9
10
11 echo "<div id=\"". $tainted ."\">content</div>" ;
12 ?>
13 <h1>Hello World!</h1>
14 </body>
15 </html>
```

Listing 5.4: Test case 191378, marked as ”safe” by SARD and ”unsafe” by us.

false and result in the assignment of the value `'safe2'` to `$tainted`.

Undefined constant: Some of SARD's test cases contain an undefined constant named `checked_data` in their sensitive sinks (line 11 in Listing 5.6). We believe this to be a mistake, but decided to include these test cases in our evaluation because they are part of the original dataset. The test cases in this condition are marked as ”unsafe” by SARD but we marked them as ”safe” because their input is never included in the output.

Inclusion of input inside quoted CSS property values: In some test cases, the input is included inside of quoted CSS property values (line 11 in Listing 5.7). SARD lists these test cases as ”unsafe”, but we marked them as ”safe” due to the fact that we could not find any malicious input that lead to an attack in the most popular web browsers, including in some older versions of one of them.

Inclusion of input inside of a HTML comment: Some test cases include their input inside of a HTML comment, after sanitizing it with a function that encodes special HTML characters. In Listing 5.8, note that a HTML comment is opened in line 4 (via the `<!--` characters) and closed in line 13 (via the `-->` characters). This means that any output produced by the execution of PHP between these two lines is ignored by the web browser. In these test cases, the < and > characters are encoded, preventing any malicious input from closing the HTML comment and executing malicious code. SARD lists these test cases as ”unsafe” but we marked them as ”safe”.

```
<!DOCTYPE html>
<html>
<head/>
<body>
<div>
<?php
$tainted = $_GET['UserData'];

$tainted = $tainted  == 'safe1' ? 'safe1' : 'safe2';

//flaw
echo $tainted ;
?>
</div>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.5: Test case 191564, marked as ”unsafe” by SARD and ”safe” by us.

```
<!DOCTYPE html>
<html>
<head/>
<body>
<?php
$tainted = $_GET['UserData'];

//no_sanitizing

//flaw
echo "<span style=\"color :". checked_data ."\">Hey</span>" ;
?>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.6: Test case 191410, marked as ”unsafe” by SARD and ”safe” by us.

```
<!DOCTYPE html>
<html>
<head>
<style>
<?php
$tainted = $_POST['UserData'];

$tainted = htmlentities($tainted, ENT_QUOTES);

//flaw
echo "body { color :\"". $tainted ."\" ; }" ;
?>
</style>
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.7: Test case 192710, marked as ”unsafe” by SARD and ”safe” by us.

```
<!DOCTYPE html>
<html>
<head>
<!--
<?php
$tainted = $_GET['UserData'];

$tainted = htmlspecialchars($tainted, ENT_QUOTES);

//flaw
echo $tainted ;
?>
-->
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.8: Test case 191454, marked as ”unsafe” by SARD and ”safe” by us.

		Actual Status		Total
		Vulnerable	Not Vulnerable	
Tool Status	Vulnerable	308	172	480
	Not Vulnerable	112	1172	1284
Total		420	1344	1764

Table 5.5: Summary of the tool's results for the XSS test cases.

5.2 XSS Evaluation With SARD Test Cases

The tool was able to process all test cases. A summary of the results is presented in Table 5.5. It is important to note that each unsafe test case contains a single vulnerability that requires one correction. This means that the number of vulnerabilities detected by our tool is equal to the number of corrections it applied. Also note that the tool always applied one of two corrections: i) a call to the `htmlentities` function with the `ENT_QUOTES` flag, or ii) a call to the `rawurlencode` function. Firstly, we will analyze the tool's capability to detect the vulnerabilities contained in the test cases. Then, we will analyze how secure the applied corrections are.

To provide a better overview of our tool's capabilities, we calculated the following 6 common metrics from their respective formulas:

True Positive Rate (TPR): Measures the percentage of vulnerable test cases the tool identified as such. Larger is better.

$$TPR = \frac{TP}{TP + FN} \tag{5.1}$$

True Negative Rate (TNR): Measures the percentage of non-vulnerable test cases that the tool identified as such. Larger is better.

$$TNR = \frac{TN}{FP + TN} \tag{5.2}$$

False Positive Rate (FPR): Measures the percentage of non-vulnerable test cases incorrectly identified as vulnerable by the tool. Lower is better.

$$FPR = \frac{FP}{FP + TN} \tag{5.3}$$

False Negative Rate (FNR): Measures the percentage of vulnerable test cases missed by the tool. Lower is better.

$$FNR = \frac{FN}{FN + TP} \tag{5.4}$$

Accuracy (ACC): Measures the percentage of correct decisions made by the tool. Larger is better.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.5}$$

Metric	Value (%)
True Positive Rate (TPR)	73.3
True Negative Rate (TNR)	87.2
False Positive Rate (FPR)	12.8
False Negative Rate (FNR)	26.7
Accuracy (ACC)	83.9
Precision (P)	64.2

Table 5.6: Summary of the calculated metrics for XSS.

Precision (P): Measures the percentage of vulnerable test cases correctly identified by the tool. Larger is better.

$$P = \frac{TP}{TP + FP} \tag{5.6}$$

In the formulas above, we used abbreviations for the following expressions: True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN). The results of each of the above metrics are summarized in Table 5.6.

After calculating these metrics, we analyzed the false positives and negatives in more detail to uncover the reasons that caused them. We encountered five reasons that lead to false negatives and two reasons that lead to false positives, that we describe next.

5.2.1 Reasons for False Negatives

Regarding false negatives, they all occurred for test cases that use sanitization methods that encode HTML's special characters and are regarded as safe by our tool. Their causes are described next in greater detail and assigned abbreviations for later reference:

FNRe1 - Inclusion of input inside unquoted attributes: These false negatives occurred for test cases that include the input inside of an unquoted HTML attribute. This makes the test cases vulnerable because it is possible to write nonexistent Javascript event handlers without using HTML's special characters. The vulnerability contained in the following test case (shown in Listing 5.9) can be triggered by providing `1 onmouseover=alert(1)` as input.

FNRe2 - Inclusion of input inside CSS: These false negatives occurred for test cases that include their input inside of a `<style>` tag. This makes them vulnerable because certain versions of some browsers allow the execution of some Javascript statements inside of CSS. The example given in Listing 5.10 can be attacked in some browsers by providing it with the input `expression(alert(1))`.

FNRe3 - Inclusion of input inside of a script tag: False negatives also occurred for test cases that include their input inside of a `<script>` tag. This makes them vulnerable because it is possible to write meaningful Javascript without using HTML's

```
1  <!DOCTYPE html>
2  <html>
3  <head/>
4  <body>
5  <?php
6  $tainted = $_GET['UserData'];
7
8  $tainted = htmlspecialchars($tainted, ENT_QUOTES);
9
10 //flaw
11 echo "<div id=". $tainted .">content</div>" ;
12 ?>
13 <h1>Hello World!</h1>
14 </body>
15 </html>
```

Listing 5.9: Test case 191460, vulnerable despite the use of safe sanitization.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <style>
5  <?php
6  $tainted = $_POST['UserData'];
7
8  $tainted = htmlentities($tainted, ENT_QUOTES);
9
10 //flaw
11 echo "body { color :". $tainted ." ; }" ;
12 ?>
13  </style>
14  </script>
15 </head>
16 <body>
17 <h1>Hello World!</h1>
18 </body>
19 </html>
```

Listing 5.10: Test case 192709, vulnerable despite the use of safe sanitization.

```
<!DOCTYPE html>
<html>
<head>
<script>
<?php
$tainted = $_POST['UserData'];

$tainted = htmlentities($tainted, ENT_QUOTES);

//flaw
echo $tainted ;
?>
</script>
</head>
<body onload="xss()">
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.11: Test case 192692, vulnerable despite the use of safe sanitization.

special characters. In the example provided in Listing 5.11, an attacker can simply provide the input `alert(1)` to trigger the vulnerability.

FNRe4 - Inclusion of input in a HTML tag name: These false negatives occurred for test cases that include their input in the place of a HTML tag name. Similarly to the first reason, an attacker can craft a malicious input that adds nonexistent Javascript event handlers without using HTML's special characters. An example of input that can trigger the vulnerability in Listing 5.12 is `a onmouseover=alert(1).`

FNRe5 - Inclusion of input in a HTML attribute name: This reason is very similar to the previous one, except that the test case's input is included in the place of a HTML attribute name. To attack the example (in Listing 5.13), an attacker can provide the input `onmouseover=alert(1) id`.

The number of test cases that occurred for each of the False Negative Reasons (FNRe) is summarized in Table 5.7. We believe the false negatives that occurred for FNRe2 and FNRe3 could be avoided if the tool's taint analysis had the capability to keep track of the type of sanitization function applied to a variable. This would allow it to know that, for example, a variable is not safe to include inside of a `<script>` tag if it was sanitized by a HTML-encoding function. As for the remaining FNRes, avoiding them would require a detailed analysis of the context in which a tainted variable is included in the output.

```
<!DOCTYPE html>
<html>
<head/>
<body>
<?php
$tainted = $_GET['UserData'];

$tainted = htmlspecialchars($tainted, ENT_QUOTES);

//flaw
echo "<".  $tainted ." href= \"/bob\" />" ;
?>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.12: Test case 191456, vulnerable despite the use of safe sanitization.

```
<!DOCTYPE html>
<html>
<body>
<?php
$tainted = $_GET['UserData'];

$tainted = htmlspecialchars($tainted, ENT_QUOTES);

//flaw
echo "<div ". $tainted ."= bob />" ;
?>
<h1>Hello World!</h1>
</div>
</body>
</html>
```

Listing 5.13: Test case 191455, vulnerable despite the use of safe sanitization.

Reason	Test Cases
FNRe1 - Inclusion of input inside unquoted attributes	16
FNRe2 - Inclusion of input inside CSS	32
FNRe3 - Inclusion of input inside of a script tag	32
FNRe4 - Inclusion of input in a HTML tag name	16
FNRe5 - Inclusion of input in a HTML attribute name	16
Total	112

Table 5.7: Number of false negatives for each of the presented reasons.

```
<!DOCTYPE html>
<html>
<head>
<script>
<?php
$tainted = $_POST['UserData'];

$tainted = addslashes($tainted);


echo "x='". $tainted ."'" ;
?>
</script>
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.14: Test case 192641, safe because the input is included in a Javascript string after sanitizing quotes.

5.2.2 Reasons for False Positives

Regarding false positives, they occurred for test cases that use unsafe sanitization methods in a context that makes them safe and for test cases that use a sanitization method involving a regular expression. Their causes are described next in greater detail and assigned abbreviations for later reference:

FPRe1 - Unsafe sanitization used in a context that makes it safe: These false positives occurred for test cases that sanitize quotes. The inclusion of input inside of a Javascript string or a quoted CSS property value is safe in these cases because any quotes present in the input are sanitized by preceding them with backslashes. This means that an attacker can not execute meaningful code. Listing 5.14 shows an example of a test case in this situation.

FPRe2 - Use of a sanitization method that involves a regular expression: All the false positives that occurred for this reason were caused by calls to `preg_replace` with a safe regular expression. Our tool does not currently handle regular expressions, meaning that any calls to `preg_replace` are considered to return tainted data, regardless of the regular expression used. Listing 5.15 shows an example of a test case in this situation.

The number of test cases that occurred for each of the False Positive Reasons (FPRe) presented is summarized in Table 5.8.

```
<!DOCTYPE html>
<html>
<head/>
<body>
<div>
<?php
$tainted = $_GET['UserData'];

$tainted = preg_replace('/\W/si','',$tainted);


echo $tainted ;
?>
</div>
<h1>Hello World!</h1>
</body>
</html>
```

Listing 5.15: Test case 191543, safe because the sanitization removes any character that is not a letter or a number.

Reason	Test Cases
FPRe1 - Unsafe sanitization used in a context that makes it safe	72
FPRe2 - Use of a sanitization method that involves a regular expression	100
Total	172

Table 5.8: Number of false positives for each of the presented reasons.

Group	Corrections
Unneeded	172
Safe	228
Unsafe	80
Total	480

Table 5.9: Summary of the applied XSS corrections.

5.2.3 Applied XSS Corrections

With regard to the XSS corrections applied by the tool, we manually analyzed all of them to determine how many of them actually prevent attacks. It is important to note that none of the corrections caused a program to become syntactically invalid. To complete this task, we looked at the location where the input is included in the program's output to determine whether the correction can prevent all attacks. For example, if the input is included in the place of a HTML tag name, our tool's HTML-encoding correction can not prevent all attacks. Given our knowledge about the test cases containted in our dataset and the corrections applied by the tool, we were able to complete this task by looking at the file names in our dataset, due to the fact that the SARD's file names follow a well-defined structure. We grouped the corrections into the following three groups:

Unneeded: These corrections were applied to non-vulnerable files meaning that they were not necessary. Note that these corrections were applied to the test cases that resulted in false positives for the reasons we presented in Subsection 5.2.2.

Safe: These corrections prevent all attacks, making the programs safe.

Unsafe: These corrections do not prevent all attacks, leaving the programs vulnerable.

Table 5.9 shows the number of corrections in each of the groups.

5.3 SQLi Evaluation With SARD Test Cases

As with XSS, the tool was able to process all test cases and a summary of the results is presented in Table 5.10. As with XSS, each unsafe test case contains a single vulnerability that requires one correction. This means that the number of vulnerabilities detected by our tool is equal to the number of corrections it applied. The tool always applied the same correction: a call to the `mysql_real_escape_string` function. Firstly, we will analyze the tool's capability to detect the vulnerabilities. Then, we will analyze how secure the applied corrections are.

To provide a better overview of our tool's capabilities, we calculated the same 6 metrics that we calculated for XSS. The results of these metrics are presented in Table 5.11.

		Actual Status		Total
		Vulnerable	Not Vulnerable	
Tool Status	Vulnerable	10	13	23
	Not Vulnerable	3	74	77
Total		13	87	100

Table 5.10: Summary of the tool's results for the SQLi test cases.

Metric	Value (%)
True Positive Rate (TPR)	76.9
True Negative Rate (TNR)	85.1
False Positive Rate (FPR)	14.9
False Negative Rate (FNR)	23.1
Accuracy (ACC)	84.0
Precision (P)	43.5

Table 5.11: Summary of the calculated metrics for SQLi.

After calculating these metrics, we analyzed the false positives and negatives in more detail, to uncover the reasons that caused them. There were four reasons that lead to false positives and a single reason that lead to false negatives.

5.3.1 Reason for False Negatives

Three false negatives occurred in this part of the evaluation and they were all caused by the use of the `mysql_real_escape_string` function to sanitize data that is later included in a comparison with an integer. This corresponds to the problem that was explained in Subsection 2.3.2. Listing 5.16 provides an example of one of the false negatives.

Avoiding these false negatives would require a more detailed analysis of the query's structure to determine the column's data type.

5.3.2 Reasons for False Positives

Regarding false positives, they occurred for varying reasons that will be described next, and assigned abbreviations for later reference.

FPRe1 - Use of a sanitization method that escapes quotes: Two false positives occurred due to the use of addslashes (or an equivalent filter) to sanitize the input before it is included in the query. This type of sanitization is regarded as unsafe by our tool but it is safe in these test cases because any quotes present in the input are sanitized by preceding them with backslashes. Listing 5.17 presents an example of one of these test cases.

FPRe2 - Use of a XSS sanitization method: Seven false positives occurred due to the use of a XSS sanitization method to sanitize the input. This is safe in these test

```
1  <?php
2  $tainted = $_POST['UserData'];
3
4  $tainted = mysql_real_escape_string($tainted);
5
6  $query = "SELECT Trim(a.FirstName) & ' ' & Trim(a.LastName) AS
       employee_name, a.city, a.street & (' ' +a.housenum) AS
       address FROM Employees AS a  WHERE a.supervisor=". $tainted .
        "";
7
8  //flaw
9  $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
       '); // Connection to the database (address, user, password)
10 mysql_select_db('dbname') ;
11 echo "query : ". $query ."<br /><br />" ;
12
13 $res = mysql_query($query); //execution
14
15 while($data =mysql_fetch_array($res)){
16 print_r($data) ;
17 echo "<br />" ;
18 }
19 mysql_close($conn);
20
21 ?>
```

Listing 5.16: Test case 167180, vulnerable despite the use of a sanitization function.

```php
<?php
$array = array();
$array[] = 'safe' ;
$array[] = $_GET['userData'] ;
$array[] = 'safe' ;
$tainted = $array[1] ;

$tainted = addslashes($tainted);

$query = sprintf("SELECT * FROM '%s'", $tainted);

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
    '); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Listing 5.17: Test case 183625, safe because the input is included inside a SQL string after sanitizing quotes.

```php
<?php

$tainted = $_GET['UserData'];

$tainted = htmlspecialchars($tainted, ENT_QUOTES);

$query = "SELECT * FROM '". $tainted . "'";

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
    '); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Listing 5.18: Test case 163730, safe because the input is included inside a SQL string after sanitizing it with a HTML-encoding function.

cases. However, it is not considered safe by our tool because XSS sanitization functions should never be used to prevent SQLi. Listing 5.18 presents an example of one of these test cases.

FPRe3 - Use of a sanitization method involving a regular expression: Two false positives occurred due to a call to `preg_replace` with a safe regular expression. As mentioned in Subsection 5.2.2, our tool considers calls to `preg_replace` to return tainted data. Listing 5.19 shows an example of a test case in this situation.

FPRe4 - Use of a numeric format specifier: Two false positives occurred due to the use of a numeric format specifier in a call to `sprintf`. This effectively consists of casting the input to a numeric type, which was described in Subsection 2.2.1. Our tool currently does not support calls to `sprintf`, meaning that the format specifiers are not taken into consideration during the taint analysis. Listing 5.20 contains an example of a test case in this situation. Note the use of `%u` to format the input as an unsigned decimal number.

We believe that the false positives that occurred for reasons FPRe1 and FPRe2 can not be avoided because these two methods of sanitization should not be considered safe for SQLi. As for the false positives that occurred for the remaining reasons, avoiding them

```php
<?php

$tainted = $_GET['UserData'];

$tainted = preg_replace('/\W/si','',$tainted);

$query = sprintf("SELECT * FROM '%s'", $tainted);

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
    '); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Listing 5.19: Test case 163918, safe because the input is sanitized with a safe regular expression.

```php
<?php

$tainted = $_POST['UserData'];

//no_sanitizing

$query = sprintf("SELECT * FROM COURSE c WHERE c.id IN (SELECT
   idcourse FROM REGISTRATION WHERE idstudent='%u')", $tainted);

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password
   '); // Connection to the database (address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);

?>
```

Listing 5.20: Test case 166930, safe because a numeric format specifier is used to include the input in the query.

Group	Corrections
Unneeded	13
Safe	9
Unsafe	1
Total	23

Table 5.12: Summary of the applied SQLi corrections.

would require a better analysis of regular expressions or the simulation of the execution of calls to `sprintf`.

5.3.3 Applied SQLi Corrections

With regard to the corrections applied by the tool to the SQLi test cases, we analyzed all of them to determine whether they prevent all attacks. It must be noted that, as with XSS, none of the applied corrections caused a program to become syntactically invalid. To complete this task, we manually analyzed all corrected programs (23 in total) and determined the safety of the correction applied to each of them. As with XSS, we grouped the corrections into the following three groups:

Unneeded: These corrections were applied to non-vulnerable files meaning that they were not necessary. Note that these corrections were applied to the test cases that resulted in false positives for the reasons we presented in Subsection 5.3.2.

Safe: These corrections prevent all attacks, making the programs safe.

Unsafe: These corrections do not prevent all attacks, leaving the programs vulnerable.

Table 5.12 shows the number of corrections in each of the groups.

5.4 Real Web Applications

SARD test cases allowed the evaluation of the tool's capability to deal with diverse sanitization methods. However, SARD's test cases contain artificial code which might not represent real web applications accurately. In order to test our tool with real code, we obtained six web applications that were vulnerable to XSS from the site Exploit-DB.

Before describing our work for this part of the evaluation, it is worth mentioning that most of the files contained in these applications do not fit our definition of a slice of code because they contain, for example, if statements. For this reason, our tool can not be expected to perform a complete and detailed analysis. Also, our tool analyzes each file in isolation, that is, it does not take other included files into consideration. These two factors may have influenced the number of vulnerabilities found for these applications.

Application	Vulnerable Version	Latest Version	PHP Files	PHP LoC	Type of Application
Site@School	2.4.10		567	64 k	Content Management System for Primary Schools
Integria IMS	5.0.83	5.0.85	974	198 k	IT Service Support Management Tool
Electricks eCommerce	1.0		45	7 k	E-Commerce Website
userSpice	4.3	4.4	474	114 k	User Management Application
AShop Shopping Cart	6.0.2		628	113 k	Shopping Cart Software
I, Librarian	4.6	4.10	114	26 k	PDF File Manager

Table 5.13: Description of the applications used in our evaluation.

5.4.1 Application Description

Table 5.13 presents a summary of the applications that were collected. The table includes a description of the application's type, vulnerable version, latest version as of June 2019, number of PHP files contained in it and number of PHP lines of code (LoC). The LoC counts were computed using the cloc[4] tool. The number of PHP files presented in the table is the same as the number of files that our tool analyzes for each application.

Considering XSS, these six applications contain a mixture of sensitive sinks considered by our tool, such as `echo`, `print`, `die` and `exit`. This showed that our tool is capable of detecting sensitive sinks other than `echo`, which is the only sensitive sink in the test cases from SARD.

5.4.2 Results

Among the six applications, the tool considered a total of 38 PHP files to be vulnerable with a total of 79 variables requiring correction. The resulting corrected programs are all syntactically valid. All but one of the resulting programs preserved their functionality. Table 5.14 presents a summary of the files corrected and number of corrections applied for each of the applications being considered.

The tool applied 72 corrections using HTML-encoding functions and 7 corrections using URL-encoding functions. On 77 occasions, the correction was applied to the tainted variable itself and, on 2 occasions, the correction was carried out on a variable's taint causes because the variable contained HTML tags in it's simulated value. The functionality of the applications was only affected with one of the corrections applied to the I, Librarian application. The program whose functionality was affected by our correction is shown in Listing 5.21. In this program, the whole of line 4 was added by our tool. Since

[4]https://github.com/AlDanial/cloc

Application	Vulnerable Files	Corrections Applied
Site@School	13	16
Integria IMS	5	5
Electricks eCommerce	5	27
userSpice	1	1
AShop Shopping Cart	8	24
I, Librarian	6	6
Total	38	79

Table 5.14: Summary of the files corrected for each of the applications.

Application	Safe Corrections	Unsafe Corrections
Site@School	16	0
Integria IMS	5	0
Electricks eCommerce	23	4
userSpice	1	0
AShop Shopping Cart	24	0
I, Librarian	6	0
Total	75	4

Table 5.15: Safety of the corrections applied to the real applications.

the if statement is written without curly braces, the addition of our correction caused the `die` to be ”moved outside” of the if statement, meaning that it will always be executed, unlike what the developer originally intended. In PHP, when an if statement is written without curly braces, only the first instruction after the condition is considered to be part of the statement. Note that this file contains if statements, meaning that it does not fit our definition of a slice of code.

```
1 <?php
2 $title = trim($_POST['title']);
3     if (empty($error))
4 $title = htmlentities($title, ENT_QUOTES);
5         die('title:' . lib_htmlspecialchars($title));
```

Listing 5.21: Code excerpt taken from I, Librarian (simplified for readability).

As shown in Table 5.15, 75 of the 79 corrections are safe, preventing all attacks. Four corrections applied to Electricks eCommerce are unsafe because, in those situations, the input is included inside of an unquoted HTML attribute, meaning that an HTML-encoding function can not prevent all attacks. Listing 5.22 provides an example of one of the unsafe corrections applied by our tool (some lines of code were broken into two for readability). The correction is the call to `htmlentities` on lines 2 and 3. Notice that the input is included in the `value` attribute on line 5 without being surrounded by any quotes, thus allowing the addition of new HTML attributes such as `onmouseover`. An example of an input that could trigger the vulnerability is `1 onmouseover=alert(1)` for the

```
1 <div class="form group">
2 <?php $_GET['prod_id'] = htmlentities($_GET['prod_id'],
3                                 ENT_QUOTES); ?>
4 <input type="hidden" class="form-control" id="prod_id"
5         name="prod_id" value=<?php echo $_GET['prod_id'];?>>
```

Listing 5.22: Correction applied to Electricks eCommerce (simplified for readability).

value of `$_GET['prod_id']`.

Chapter 6

Conclusions and Future Work

This chapter highlights the main conclusions of our work and it's limitations. It also presents some directions for future work based on the research performed.

6.1 Conclusions

In this work, we studied the different types of XSS and SQLi attacks against PHP web applications. We also analyzed a multitude of sanitization methods available in PHP for both of these vulnerabilities and the situations when they should be applied.

After studying related research efforts in the area of static analysis, we concluded that the main problem of currently available SATs is the fact that most of them do not perform automatic correction of the vulnerabilities found. The SATs that do so often produce new programs that are syntactically invalid and can not be executed. With this in mind, we proposed a solution based on taint analysis to find and correct vulnerabilities in simplified PHP programs (i.e., slices of code) by adding new lines of code containing secure corrections.

We implemented our proposed solution in the form of a static analysis tool written in Python. The developed tool was evaluated using both automatically generated test cases collected from SARD and real web applications obtained from Exploit-DB.

Regarding the tool's capability to find XSS vulnerabilities, SARD showed that the tool produces some false positives and negatives. The main cause for false positives was the use of regular expressions to sanitize input, something that is difficult to analyze due to the complexity of regular expressions. The main cause for false negatives was the location where the input is included in the output. We do not believe this to pose a big problem in practice because SARD contains artificial code that may not represent real web applications accurately.

As for the SQLi vulnerabilities, SARD showed that the tool also produces some false positives and negatives. However, approximately half of the false positives occurred due to the use of a XSS sanitization function to prevent SQLi. This is something that can not

be considered safe and prevents us from reducing the number of false positives. There were also some false positives due to the use of regular expressions to sanitize input, something that is difficult to analyze, as mentioned before. Regarding the false negatives, all of them occurred due to the location where the input is included in the query.

We did not analyze the false positives and negatives for the real web applications because the files contained in them do not fit our definition of a slice of code, a fact that may also have influenced the number of vulnerabilities found.

Regarding the tool's capability to insert corrections, the results were very satisfying. In SARD, all corrected programs were syntactically valid and preserved their original behavior for both types of vulnerabilities. In the real web applications, all corrected programs remained syntactically valid and only one of them saw it's original behavior change (the program whose behavior was affected by our tool does not fit our definition of a slice of code). Regarding the safety of the inserted corrections, the vast majority of the ones applied to the real applications was safe against all attacks. In the case of SARD there were some unsafe corrections due to the location where the input is included in the program's output or SQL query. We do not believe this to pose a big problem because SARD contains artificial code. Still in the case of SARD, the safe corrections vastly outnumber the unsafe ones for both XSS and SQLi.

To conclude, we believe that our solution fulfills the objectives outlined in Section 1.2, thus providing an advancement in the code correction capabilities of SATs.

6.2 Limitations

The main limitation of the developed solution is the fact that it's capability to correct SQLi vulnerabilities is very basic. It always adds the same type of correction, which was shown to be unsafe in some cases. This happened due to time restrictions in our work.

Another limitation of our approach is the simulation of variable operations. Although it can produce an approximate value, it does not deal well with function calls and other complex language constructs (such as class method calls). It also does not deal with calls to `sprintf`, which is a commonly used function for the construction of strings.

Another limitation of our solution is the fact that it currently does not support method calls as sensitive sinks or sanitization methods. This is not a problem in the case of XSS but it can affect the tool's capability to detect and correct more complex SQLi vulnerabilities. This happens because the MySQL Improved extension allows the use of object-oriented code to sanitize input (`mysqli::real_escape_string`) and query the database (`mysqli::query`).

6.3 Future Work

We leave the resolution of the limitations mentioned previously for future work. Another interesting direction is the extension of our proposed solution to be able to analyze complete PHP files instead of simplified ones. There is also the possibility of extending it to deal with more types of vulnerabilities and corrections.

Another interesting direction for future work is an improvement of the taint analysis to give it the capability to deal with validation functions used in predicates.

Lastly, we believe that there is room for improvements in the amount of information the tool maintains during the taint analysis. For example, the tool could keep track of the type of sanitization function used on a variable. This would help reduce the false negatives that arise for reasons such as the ones encountered in SARD.

Acronyms

ACC Accuracy.

AST Abstract Syntax Tree.

CMS Content Management System.

CSP Content Security Policy.

CSS Cascading Style Sheets.

CSV Comma-separated Values.

CWE Common Weakness Enumeration.

DOM Document Object Model.

EP Entry Point.

FN False Negatives.

FNR False Negative Rate.

FNRe False Negative Reason.

FP False Positives.

FPR False Positive Rate.

FPRe False Positive Reason.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

IETF Internet Engineering Task Force.

IT Information Technology.

LoC Lines of Code.

NIST National Institute of Standards and Technology.

OS Operating System.

OWASP Open Web Application Security Project.

P Precision.

PDF Portable Document Format.

PHP PHP: Hypertext Preprocessor.

SARD Software Assurance Reference Dataset.

SAT Static Analysis Tool.

SQL Structured Query Language.

SQLi SQL Injection.

SS Sensitive Sink.

TN True Negatives.

TNR True Negative Rate.

TP True Positives.

TPR True Positive Rate.

URL Uniform Resource Locator.

XSS Cross-Site Scripting.

Bibliography

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016.

[2] A. Algaith, P. Nunes, J. Fonseca, I. Gashi, and M. Viera. Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools. In *Proceedings of the European Dependable Computing Conference*, July 2018.

[3] D. Anderson and M. Hills. Query Construction Patterns in PHP. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, Feb 2017.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, February 2013.

[5] J. Dahse and T. Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, August 2014.

[6] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[7] L. Flynn, W. Snavely, D. Svoboda, N. VanHoudnos, R. Qin, J. Burns, D. Zubrow, R. Stoddard, and G. Marce-Santurio. Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. In *Proceedings of the International Workshop on Software Qualities and Their Dependencies*, May 2018.

[8] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, March 2016.

[9] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, November 2005.

[10] W. G. J. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*, Jan 2008.

[11] W. G. J. Halfond, J. Viegas, and A. Orso. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, March 2006.

[12] J. Huang, Y. Li, J. Zhang, and R. Dai. UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2019.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International Conference on World Wide Web*, May 2004.

[14] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium*, August 2005.

[15] I. Medeiros, N. F. Neves, and M. Correia. Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives. In *Proceedings of the International World Wide Web Conference*, April 2014.

[16] P. Nunes, I. Medeiros, J. Fonseca, N. F. Neves, M. Correia, and M. Vieira. On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *Proceedings of the European Dependable Computing Conference*, September 2017.

[17] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming*, 2018.

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[19] L. K. Shar and H. B. K. Tan. Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities. In *Proceedings of the International Conference on Software Engineering*, June 2012.

[20] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis. In *Proceedings of the International Conference on Software Engineering*, May 2013.

[21] R. Shirey. Internet Security Glossary. RFC 4949, IETF, August 2007.

[22] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, September 2015.

[23] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, April 2016.

[24] A. van der Stock, B. Glas, N. Smithline, and T. Gigler. Owasp Top 10 2017 The Ten Most Critical Web Application Security Risks. Technical report, OWASP, 2017.

[25] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the USENIX Conference on Offensive Technologies*, August 2011.

[26] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, August 2013.